Matt **Zandstra**

PHP

OBIEKTY, WZORCE I NARZĘDZIA

Wydanie V

Helion

Tytuł oryginału: PHP Objects, Patterns, and Practice, 5th Edition

Tłumaczenie: Piotr Cieślak

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-3553-0

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/phpob5.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/phpob5
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 3

Obiektowy elementarz

Obiekty i klasy są sercem tej książki, a od momentu wprowadzenia PHP5 nieco ponad dekadę temu stanowią również serce języka PHP. Niniejszy rozdział ma przygotować niezbędny grunt dla pogłębionego omówienia obiektów i projektowania obiektowego w kontekście mechanizmów języka PHP. Ci, którym programowanie obiektowe w tym języku jest całkiem obce, powinni przeczytać ten rozdział bardzo uważnie.

Omawiam w nim następujące zagadnienia:

- *Klasy i obiekty* — deklarowanie klas i tworzenie ich egzemplarzy (obiektów).
- *Konstruktory* — automatyzację konfiguracji obiektów.
- *Typy elementarne i klasy* — jakie znaczenie ma typ w PHP.
- *Dziedziczenie* — gdzie się przydaje i jak je stosować.
- *Widoczność* — udostępnianie interfejsów klas i zabezpieczanie metod i składowych obiektów przed ingerencją.

Klasy i obiekty

Pierwszą przeszkodą na drodze do zrozumienia programowania obiektowego jest dziwaczność i niezwykłość relacji pomiędzy klasą a jej obiektami. Dla wielu osób właśnie pojęcie tej relacji stanowi pierwsze olśnienie, wywołuje pierwszą ekscytację programowaniem obiektowym. Nie skąpmy więc energii na poznawanie podstaw.

Pierwsza klasa

Klasy są często opisywane w odniesieniu do obiektów. To bardzo ciekawe, ponieważ obiekty są z kolei niejednokrotnie opisywane przez pryzmat klas. Ta zależność bardzo spowalnia pierwsze postępy adeptów programowania obiektowego. Ponieważ to klasy definiują obiekty, zaczniemy od definicji klasy.

Krótko mówiąc, klasa to swego rodzaju szablon wykorzystywany do generowania jednego lub większej liczby obiektów. Deklaracja klasy zawiera słowo kluczowe `class` i dowolnie wybraną nazwę klasy. Nazwa klasy może być dowolną kombinacją cyfr i liter, nie może się jednak od cyfry zaczynać. Kod skojarzony z klasą musi być ograniczony nawiasami klamrowymi. Spróbujmy na podstawie tych informacji skonstruować klasę:

```
// listing 03.01
class ShopProduct
{
    // ciało klasy
}
```

Zdefiniowana właśnie klasa `ShopPoduct` jest prawidłową klasą, choć jej przydatność jest na razie znikoma. Mimo to osiągnęliśmy coś bardzo znaczącego, bo zdefiniowaliśmy typ. Utworzyliśmy więc kategorię danych, którą możemy wykorzystywać w skryptach. Znaczenie tego faktu stanie się niebawem jaśniejsze.

Pierwszy obiekt (lub dwa)

Skoro klasa jest szablonem generowania obiektów, obiekt stanowią dane, które zostały zaaranżowane zgodnie z szablonem definiowanym w klasie. Obiekt zwie się egzemplarzem klasy bądź jej konkretyzacją. Klasa definiuje jego typ.

Wykorzystamy teraz klasę `ShopProduct` jako formę do generowania obiektów typu `ShopProduct`. Pomoże nam w tym operator `new`. Operator `new` jest zazwyczaj stosowany w połączeniu z nazwą klasy, jak tutaj:

```
// listing 03.02
$product1 = new ShopProduct();
$product2 = new ShopProduct();
```

Operator `new`, jeśli zostanie wywoływany z nazwą klasy w roli swojego jedynego operandu, zwraca egzemplarz tejże klasy. W naszym przykładzie generuje obiekt klasy `ShopProduct`.

Klasę `ShopProduct` wykorzystaliśmy do wygenerowania dwóch obiektów typu `ShopProduct`. Choć funkcjonalnie `$product1` i `$product2` są identyczne (tzn. puste), stanowią obiekty rozłączne, a ich wspólnym mianownikiem jest jedynie wspólna klasa, na podstawie której zostały wygenerowane.

Istnienie klas i obiektów można sprowadzić do następującej analogii: klasa to jakby maszyna tłocząca gumowe kaczki. Obiektami zaś są owe gumowe kaczki produkowane przy użyciu maszyny. Ich „typ" jest wyznaczany formą, w której są odciskane. Wszystkie wypluwane z maszyny kaczki wyglądają więc identycznie, ale są niezależnymi od siebie obiektami materialnymi. Innymi słowy, są różnymi egzemplarzami pewnego przedmiotu. Aby rozróżnić poszczególne kaczki, można by im zresztą podczas wytłaczania nadawać numery seryjne. Każdy obiekt tworzony w języku PHP również posiada odrębną tożsamość, jednak unikatową jedynie w okresie życia danego obiektu (PHP ponownie wykorzystuje nieużywane już identyfikatory obiektów w obrębie tego samego procesu). Można się o tym przekonać, wydrukowując obiekty `$product1` i `$product2`:

```
// listing 03.03
var_dump($product1);
var_dump($product2);
```

Wykonanie powyższego kodu spowoduje wypisanie na wyjściu:

```
object(popp\r03\zestaw01\ShopProduct)#235 (0) {
}
object(popp\r03\zestaw01\ShopProduct)#234 (0) {
}
```

> **Uwaga** W bardzo starych wersjach PHP (do wersji 5.1 włącznie) można było wyświetlić zawartość obiektu wprost. Taka operacja powoduje zrzutowanie obiektu na ciąg znaków zawierający identyfikator obiektu. Od PHP 5.2 język został pozbawiony tej automagicznej konwersji, więc traktowanie obiektu jako ciągu znaków jest błędem, chyba że w klasie obiektu jest zdefiniowana metoda `__toString()`. Metodami zajmiemy się w dalszej części tego rozdziału, a o metodzie `__toString()` będzie mowa w rozdziale 4.

Przekazanie obiektu do wywołania `var_dump()` pozwala wypisać ciekawe dane o obiekcie, z identyfikatorem obiektu na czele (po symbolu kratki).

Aby klasę `ShopProduct` uczynić ciekawszą, możemy uzupełnić ją o obsługę specjalnych pól danych, zwanych składowymi bądź właściwościami (ang. *properties*).

Definiowanie składowych klasy

Klasy mogą definiować specjalne zmienne zwane właściwościami bądź składowymi. Składowa przechowuje dane, które różnią się pomiędzy egzemplarzami danej klasy. W przypadku obiektów klasy `ShopProduct` (niech będzie to asortyment księgarski, choć ogólnie chodzi o dowolne artykuły) możemy na przykład zażyczyć sobie obecności pól ceny (`price`) i tytułu (`title`).

Składowa klasy przypomina zwykłą zmienną, z tym że przy deklarowaniu składowej jej nazwę trzeba poprzedzić słowem kluczowym określającym widoczność. Może być nim `public`, `protected` albo `private`. Wybór słowa kluczowego widoczności określa zasięg, w którym możliwe będzie realizowanie odwołań do składowych.

■ **Uwaga** Zasięg to inaczej kontekst (klasy czy funkcji), w ramach którego zmienna posiada znaczenie (to samo dotyczy metod, o których więcej w dalszej części rozdziału). Zmienna zdefiniowana wewnątrz funkcji istnieje jedynie lokalnie, a zmienna definiowana poza funkcją istnieje globalnie. Należy przyjąć, że nie istnieje możliwość odwoływania się do danych definiowanych w zasięgu, który jest bardziej lokalny niż bieżący. Po zdefiniowaniu zmiennej wewnątrz funkcji nie można się do niej później odwołać spoza tejże funkcji. Obiekty są w tym względzie bardziej przenikalne — niektóre z ich zmiennych mogą być niekiedy dostępne z innych kontekstów. Ową dostępność reguluje się słowami `public`, `protected` i `private`.

Do kwestii widoczności i regulujących ją słów kluczowych wrócimy później. Na razie spróbujemy po prostu zadeklarować kilka składowych klasy za pomocą słowa kluczowego `public`:

```
// listing 03.04
class ShopProduct
{
    public $title             = "bez tytułu";
    public $producerMainName  = "nazwisko";
    public $producerFirstName = "imię";
    public $price             = 0;
}
```

Jak widać, uzupełniliśmy klasę o cztery składowe, przypisując do każdej z nich wartość domyślną. Wszelkie obiekty konkretyzowane na bazie takiej klasy będą teraz zawierać owe dane domyślne. Słowo kluczowe `public` poprzedzające deklarację każdej składowej umożliwia odwoływanie się do niej spoza kontekstu obiektu.

Do składowych definiowanych w obrębie klasy, a konkretyzowanych w obiektach możemy się odwoływać za pośrednictwem ciągu znaków „->” (operatora obiektów) w połączeniu ze zmienną w postaci obiektu oraz nazwą składowej:

```
// listing 03.05
$product1 = new ShopProduct();
print $product1->title;
```

```
bez tytułu
```

Ponieważ składowe zostały zdefiniowane jako publiczne (`public`), możemy odczytywać ich wartości i je do nich przypisywać, zmieniając domyślne stany obiektów definiowane w klasie:

```
// listing 03.06
$product1 = new ShopProduct();
$product2 = new ShopProduct();
$product1->title = "Moja Antonia";
$product2->title = "Paragraf 22";
```

Deklarując i ustawiając w klasie `ShopProduct` składową `$title`, wymuszamy podaną początkową wartość tej składowej we wszystkich nowo tworzonych obiektach klasy. Oznacza to, że kod użytkujący klasę może bazować na domniemaniu obecności tej składowej w każdym z obiektów klasy. Nie może już jednak domniemywać wartości składowych, gdyż te — jak wyżej — mogą się w poszczególnych obiektach różnić między sobą.

■ **Uwaga** Kod wykorzystujący klasę, funkcję bądź metodę nazwiemy *kodem klienckim* wobec tej klasy, metody czy funkcji, albo po prostu *klientem klasy* (metody, funkcji). Termin „klient" będzie się więc w tej książce pojawiał stosunkowo często.

Zresztą PHP nie zmusza nas do deklarowania wszystkich składowych w klasie. Obiekty można uzupełniać składowymi dynamicznie, jak tutaj:

```
// listing 03.07
$product1->arbitraryAddition = "nowość";
```

Taka metoda uzupełniania obiektów o składowe nie jest jednak zalecana w programowaniu obiektowym.

Czy dynamiczne uzupełnianie składowych to zła praktyka? Definiując klasę, definiuje się typ obiektów. Informuje się tym samym otoczenie, że dana klasa (i wszelkie jej konkretyzacje w postaci obiektów) składa się z ustalonego zestawu pól danych i funkcji. Jeśli klasa `ShopProduct` definiuje składową `$title`, wtedy dowolny kod użytkujący obiekty klasy `ShopProduct` może śmiało odwoływać się do składowej `$title`, ponieważ jej dostępność jest pewna. Nie da się podobnej pewności stosowania uzyskać względem składowych dodawanych do obiektów w sposób dynamiczny.

Nasze obiekty są na razie cokolwiek nieporęczne. Chcąc manipulować ich składowymi, musimy bowiem czynić to poza samymi obiektami. Sięgamy do nich jedynie celem ustawienia i odczytania składowych. Konieczność ustawienia wielu takich składowych szybko stanie się wyjątkowo uciążliwa:

```
// listing 03.08
$product1 = new ShopProduct();
$product1->title = "Moja Antonia";
$product1->producerMainName  = "Cather";
$product1->producerFirstName = "Willa";
$product1->price = 5.99;
```

W powyższym kodzie zamazaliśmy jedną po drugiej pierwotne, definiowane w klasie wartości składowych obiektów, aż wprowadziliśmy komplet pożądanych zmian obiektu. Po arbitralnym wymuszeniu wartości składowych możemy się swobodnie odwołać do nowych wartości:

```
// listing 03.09
print "Autor: {$product1->producerFirstName} "
            ."{$product1->producerMainName}\n";
```

Powyższy kod wypisze na wyjściu programu:

```
Autor: Willa Cather
```

Taka metoda ustawiania i odwoływania się do składowych powoduje szereg problemów. Największym jest potencjalne niebezpieczeństwo dynamicznego uzupełnienia zestawu składowych obiektu w wyniku literówki w odwołaniu. O taką pomyłkę naprawdę łatwo — załóżmy, że chcemy napisać tak:

```
$product1->producerMainName  = "Cather";
```

Tymczasem przez pomyłkę wpisujemy następujący kod:

```
$product1->producerSecondName = "Cather";
```

Z punktu widzenia samego języka PHP kod taki byłby jak najbardziej dozwolony, więc programista nie otrzymałby żadnego ostrzeżenia. Ale kiedy przyszłoby do wyprowadzania nazwiska autora (ogólnie: wytwórcy), wyniki odbiegałyby od oczekiwanych.

Kolejnym problemem jest zbytnie rozluźnienie relacji pomiędzy składowymi klasy. Nie mamy obowiązku ustawiać tytułu, ceny czy nazwiska autora — użytkownik obiektu może być pewien, że obiekt takie składowe posiada, ale nie ma żadnej gwarancji przypisania do nich jakichkolwiek wartości (poza ewentualnymi wartościami domyślnymi). Tymczasem najlepiej byłoby, gdyby każdy nowo utworzony obiekt `ShopProduct` posiadał znaczące wartości swoich składowych.

Wreszcie traktowanie z osobna każdej składowej jest nużące, zwłaszcza kiedy zamierzamy robić to częściej. Już samo wyświetlenie nazwiska autora jest uciążliwe.

Byłoby miło, gdyby podobne zadania dało się złożyć na barki samego obiektu.

Wszystkie te problemy można wyeliminować, uzupełniając klasę `ShopProduct` o zestaw własnych funkcji, które pośredniczyłyby w manipulowaniu składowymi, operując nimi z poziomu kontekstu obiektu.

Metody

Składowe pozwalają obiektom na przechowywanie danych, metody zaś umożliwiają obiektom wykonywanie zadań. Metody to specjalne funkcje (zwane też niekiedy funkcjami składowymi), deklarowane we wnętrzu klasy. Jak można się spodziewać, deklaracja metody przypomina deklarację zwykłej funkcji. Nazwę metody poprzedza słowo kluczowe `function`, a uzupełnia ją opcjonalna lista parametrów ujęta w nawiasy. Ciało metody ograniczone jest nawiasami klamrowymi:

```
public function myMethod($argument, $another)
{
    // ...
}
```

W przeciwieństwie do zwykłych funkcji metody muszą być deklarowane w ciele klasy. Mogą też być opatrywane szeregiem modyfikatorów, w tym słowem kluczowym określającym widoczność. Podobnie jak składowe, tak i metody można deklarować jako publiczne (`public`), chronione (`protected`) albo prywatne (`private`). Deklarując metodę jako publiczną, umożliwiamy wywoływanie jej spoza kontekstu obiektu. Pominięcie określenia widoczności w deklaracji metody oznacza niejawnie jej widoczność i dostępność publiczną. Za dobrą praktykę uważa się jednak jawne deklarowanie widoczności dla wszystkich metod (do modyfikatorów metod wrócimy w dalszej części tego rozdziału).

// listing 03.10

```
class ShopProduct
{
    public $title             = "bez tytułu";
    public $producerMainName  = "nazwisko";
    public $producerFirstName = "imię";
    public $price             = 0;

    public function getProducer()
    {
        return $this->producerFirstName . " "
               . $this->producerMainName;
    }
}
```

W większości przypadków metoda wywoływana jest na rzecz konkretnego obiektu, z którym jej nazwa jest kojarzona operatorem dostępu do składowej `->`. Nazwa metody musi być w wywołaniu uzupełniona nawiasami — niezależnie od tego, czy metoda przyjmuje jakiekolwiek argumenty (dokładnie tak jak w funkcji).

// listing 03.11

```
$product1 = new ShopProduct();
$product1->title = "Moja Antonia";
$product1->producerMainName = "Cather";
$product1->producerFirstName = "Willa";
$product1->price = 5.99;

print "Autor: {$product1->getProducer()}\n";
```

Na wyjściu programu uzyskamy:

```
Autor: Willa Cather
```

Do klasy `ShopProduct` dodaliśmy metodę `getProducer()`. Warto zauważyć, że metoda `getProducer()` została zadeklarowana jako publiczna, co oznacza, że da się ją wywołać spoza wspomnianej klasy.

W ciele niniejszej metody pojawiła się pewna nowinka. Chodzi o pseudozmienną `$this`, za pośrednictwem której kod klasy odwołuje się do egzemplarza klasy, na rzecz którego metoda została wywołana. Wszelkie wątpliwości co do znaczenia `$this` w kodzie klasy należy rozstrzygać, zastępując zmienną wyrażeniem „bieżący egzemplarz klasy". Stąd instrukcja:

```
$this->producerFirstName
```

oznacza tyle, co:

```
Składowa $producerFirstName bieżącego egzemplarza klasy
```

Jak widać, metoda `getProducer()` realizuje i zwraca konkatenację składowych `$producerFirstName` i `$producerMainName`. Obecność tej metody oszczędza nam odwołań do poszczególnych składowych i własnoręcznego konstruowania ciągu nazwiska autora.

Tym sposobem ulepszyliśmy nieco naszą klasę. Mimo to nie uniknęliśmy pułapki nadmiernej elastyczności — inicjalizację obiektów klasy `ShopProducer` składamy bowiem na barki programisty kodu klienckiego klasy `ShopProduct` i musimy polegać na jego solidności. Poprawna i pełna inicjalizacja obiektu naszej klasy wymaga pięciu wierszy kodu (pięciu instrukcji) — żaden programista nam za to nie podziękuje. A do tego jako twórcy klasy nie mamy możliwości zagwarantowania prawidłowej inicjalizacji którejkolwiek ze składowych obiektów klasy `ShopProduct` w kodzie klienckim. Potrzebowalibyśmy do tego metody wywoływanej automatycznie w przebiegu konkretyzacji obiektu.

Metoda konstrukcji obiektu

Metoda konstrukcji obiektu, zwana po prostu konstruktorem, wywoływana jest w ramach konkretyzacji, czyli tworzenia obiektu klasy. W jej ramach można wykonać operacje inicjalizujące obiekt oraz wykonujące pewne przewidziane dla całej klasy operacje wstępne.

Uwaga W wersjach PHP poprzedzających wersję piątą konstruktor przyjmował nazwę klasy, w ramach której operował — klasa `ShopProduct` miała więc zawsze konstruktor `ShopProduct()`. To rozwiązanie nie działa już we wszystkich przypadkach, a w PHP7 jest uznawane za przestarzałe. Nadawaj konstruktorowi nazwę w postaci `__construct()`.

Zauważmy, że nazwa konstruktora zaczyna się od dwóch znaków podkreślenia, charakterystycznych również dla wielu innych specjalnych metod deklarowanych w klasach PHP. Zdefiniujmy więc konstruktor klasy `ShopProduct`:

```
// listing 03.12
class ShopProduct
{
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price = 0;

    public function __construct(
        $title,
        $firstName,
        $mainName,
        $price
    ) {
        $this->title = $title;
        $this->producerFirstName = $firstName;
```

```
        $this->producerMainName = $mainName;
        $this->price = $price;
    }
    public function getProducer()
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
}
```

Znów ulepszyliśmy nieco klasę, oszczędzając sobie i innym użytkownikom klasy konieczności zwielokrotniania kodu inicjalizacji każdego obiektu z osobna. Teraz przy okazji konkretyzacji obiektu w ramach operatora `new` wywoływana jest każdorazowo metoda `__construct()`. Obiekt tworzy się teraz tak:

```
// listing 03.13
$product1 = new ShopProduct(
    "Moja Antonia",
    "Willa",
    "Cather",
    5.99
);
print "Autor: {$product1->getProducer()}\n";
```

Na wyjściu tego programu otrzymamy:

```
Autor: Willa Cather
```

Wszelkie argumenty przekazane w wywołaniu `new` są przekazywane do konstruktora klasy. W naszym przykładzie przekazujemy w ten sposób do konstruktora tytuł, imię i nazwisko autora oraz cenę książki. Konstruktor w swoim ciele odwołuje się do składowych tworzonego obiektu za pośrednictwem pseudozmiennej `$this`.

■ **Uwaga** Obiekty klasy `ShopProduct` dają się teraz tworzyć znacznie łatwiej i bezpieczniej. Całość operacji związanych z inicjalizacją realizuje z punktu widzenia użytkownika pojedyncze wywołanie operatora `new`. Teraz w kodzie wykorzystującym obiekty klasy `ShopProduct` można w pełni polegać na prawidłowej inicjalizacji wszystkich składowych obiektu.

Tego rodzaju pewność i przewidywalność to bardzo istotny aspekt programowania obiektowego. Klasy należy projektować tak, aby ich użytkownicy mogli w sposób pewny wykorzystywać ich cechy. Jeden ze sposobów na poprawienie bezpieczeństwa stosowania obiektu polega na zadeklarowaniu w przewidywalny sposób typów danych przechowywanych we właściwościach (składowych) tego obiektu. Można na przykład zagwarantować, by składowa `$name` zawsze zawierała tylko dane znakowe. Tylko jak to osiągnąć, gdy dane składowej są przekazywane poza obręb klasy? Mechanizmom wymuszania konkretnego typu obiektów w deklaracjach metod przyjrzymy się w następnym podrozdziale.

Typy argumentów metod

Typy określają w skryptach sposób zarządzania danymi. Typy łańcuchowe są wykorzystywane do przechowywania i wyświetlania ciągów znaków oraz do manipulowania takimi ciągami za pośrednictwem odpowiednich funkcji. Zmienne liczbowe są wykorzystywane w wyrażeniach matematycznych. Zmienne logiczne osadzane są w wyrażeniach logicznych. Tego rodzaju typy zaliczamy do typów elementarnych (ang. *primitive types*). Klasy stanowią w systemie typów znacznie wyższy poziom. Obiekt klasy `ShopProduct` stanowi wartość elementarnego typu „obiekt”, ale równocześnie jest wcieleniem (egzemplarzem) konkretnej klasy — `ShopProduct`. Zajmijmy się więc relacjami typów i metod.

Definicje metod i funkcji nie muszą nakładać na parametry żadnych ograniczeń co do typów. To zarówno możliwość zbawienna, jak i katastrofalna. Fakt, że argument wywołania funkcji może być dowolnego typu, daje niezrównaną elastyczność. Można dzięki temu konstruować metody reagujące inteligentnie na różne przekazywane

do nich dane, dostosowując realizowane w nich funkcje do okoliczności wywołania. Elastyczność ta jest jednak równocześnie przyczyną niejednoznaczności i nieoczekiwanego działania kodu, kiedy przekazany argument jest typu innego niż spodziewany.

Typy elementarne

PHP to język o osłabionej kontroli typów. Oznacza to, że deklaracja zmiennej nie musi określać i narzucać jej typu. Zmienna `$number` może w ramach jednego zasięgu zostać zainicjalizowana wartością liczbową 2, a za chwilę nadpisana ciągiem `"dwa"`. W językach ze ścisłą kontrolą typów, jak C i Java, typ zmiennej musi być zadeklarowany jeszcze przed przypisaniem jej wartości, a przypisywana wartość musi być typu zgodnego z deklarowanym.

Nie oznacza to, że w języku PHP w ogóle nie istnieje pojęcie typu. Każda wartość, którą da się przypisać do zmiennej, posiada typ. Typ zmiennej można określić za pośrednictwem jednej z wbudowanych funkcji PHP. Lista typów elementarnych wyróżnianych w PHP wraz z funkcjami wykrywającymi przynależność do tegoż typu widnieje w tabeli 3.1. Każda z tych funkcji przyjmuje w wywołaniu badaną zmienną i zwraca wartość `true`, jeśli zmienna ta należy do określonego typu.

Tabela 3.1. *Typy elementarne i funkcje kontroli typów w PHP*

Funkcja testująca przynależność do typu	Nazwa typu	Opis
`is_bool()`	`boolean`	Jedna z dwóch wyróżnionych wartości: `true` („prawda”) i `false` („fałsz”).
`is_integer()`	`integer`	Liczba całkowita (równoważne z wywołaniami `is_int()` i `is_long()`).
`is_double()`	`double`	Liczba zmiennoprzecinkowa (z częścią ułamkową; równoważne wywołaniu `is_float()`).
`is_string()`	`string`	Ciągi znaków.
`is_object()`	`object`	Obiekt.
`is_array()`	`array`	Tablica.
`is_resource()`	`resource`	Uchwyt identyfikujący i pośredniczący w komunikacji z zasobami zewnętrznymi, np. bazami danych i plikami.
`is_null()`	`null`	Wartość pusta.

Sprawdzanie typu wartości w PHP ma szczególne znaczenie przy przetwarzaniu argumentów wywołania funkcji i metod.

Znaczenie typu elementarnego — przykład

W kodzie trzeba bezwzględnie kontrolować wykorzystywane typy. Spójrzmy na przykład jednego z wielu problemów związanych z systemem typów.

Wyobraźmy sobie, że skrypt wyodrębnia konfigurację aplikacji z pliku XML. Element XML `<resolvedomains>` instruuje aplikację co do podejmowania próby odwzorowania adresu IP na nazwę domenową — często odwzorowanie takie jest przydatne, ale zazwyczaj jest operacją stosunkowo skomplikowaną. Oto próbka pliku konfiguracyjnego:

```
<!-- listing 03.14 -->
<settings>
    <resolvedomains>false</resolvedomains>
</settings>
```

Skrypt wyodrębnia z pliku konfiguracyjnego ciąg `"false"` i przekazuje go w roli znacznika do metody o nazwie `outputAddresses()`, wyświetlającej dane adresowe (IP i ewentualnie — w zależności od wartości znacznika — nazwę domenową). Oto kod metody `outputAddresses()`:

```
// listing 03.15
class AddressManager
{
    private $addresses = ["209.131.36.159", "216.58.213.174"];
    public function outputAddresses($resolve)
    {
        foreach ($this->addresses as $address) {
            print $address;
            if ($resolve) {
                print " (".gethostbyaddr($address).")";
            }
            print "\n";
        }
    }
}
```

Klasa `AddressManager` mogłaby oczywiście zostać nieco ulepszona; wpisywanie na sztywno adresu IP w kodzie klasy rzadko kiedy jest dobrym pomysłem. Tak czy inaczej metoda `outputAddresses()` przegląda tablicę ze składowej `$addresses` i wypisuje wartości poszczególnych elementów tablicy. Jeśli parametr `$resolve` ma wartość `true`, obok adresów IP wyprowadzane są nazwy domenowe.

Oto nieco inne podejście, z wykorzystaniem w klasie `AddressManager` pliku konfiguracyjnego w formacie XML. Zobaczmy, czy uda się nam wychwycić słabość tego wariantu:

```
// listing 03.16
$settings = simplexml:load_file(__DIR__."/resolve.xml");
$manager = new AddressManager();
$manager->outputAddresses((string)$settings->resolvedomains);
```

Celem wyodrębnienia z pliku ustawień wartości elementu `resolvedomains` odwołujemy się tu do SimpleXML API. Wiemy skądinąd, że wartością owego elementu jest u nas ciąg znaków `"false"` i zgodnie z dokumentacją SimpleXML rzutujemy tę wartość na typ `string`.

Kod, niestety, nie będzie zachowywał się prawidłowo. Otóż przekazując w wywołaniu metody `outputAddresses()` ciąg `"false"`, wykazujemy się niezrozumieniem niejawnego założenia, jakie metoda czyni odnośnie do wartości argumentu wywołania. Otóż metoda spodziewa się przekazania wartości logicznej (czyli wartości `true` albo `false`). Tymczasem ciąg `"false"` nie jest wartością logiczną, a co gorsza, jeśli już użyjemy go w roli takiej wartości, da wartość `true`. PHP wykona bowiem rzutowanie niepustego ciągu znaków na typ logiczny, a w dziedzinie wartości typu logicznego niepusty ciąg znaków reprezentowany jest jako `true`. Dlatego następujący kod:

```
if ("false") {
    // ...
}
```

jest w istocie równoznaczny z:

```
if (true) {
    // ...
}
```

Błąd tego rodzaju można wyeliminować na kilka sposobów.

Można metodę `outputAddresses()` uodpornić na mylne interpretacje typów argumentów, wyposażając ją w kod rozpoznający argument typu ciągu znaków i konwertujący taki ciąg na wartość logiczną wedle własnych kryteriów:

```
// listing 03.17
    public function outputAddresses($resolve)
    {
        if (is_string($resolve)) {
            $resolve =
               (preg_match("/^(false|no|off)$/i", $resolve) ) ? false : true
        }
```

```
        // ...
    }
```

Istnieją jednak solidne przesłanki do unikania takich sposobów. W zasadzie lepiej jest udostępnić przejrzysty, zwarty i ograniczony interfejs metody niż interfejs otwarty i wieloznaczny. Funkcje i metody przyjmujące niejasne semantycznie argumenty prowokują bowiem do niechlujnego stosowania, a więc i do wprowadzania błędów użycia.

Można jeszcze inaczej: zostawić ciało metody `outputAddresses()` w spokoju, opatrując jej deklarację komentarzem dającym użytkownikom jasność co do wymagań metody wobec typu argumentu `$resolve` i jego interpretacji w ciele funkcji. Decydujemy się tym samym na złożenie odpowiedzialności za poprawne działanie metody na barki użytkownika.

```
/**
 * Wyświetla listę adresów.
 * Przy wartości true argumentu $resolve adresy będą odwzorowywane do nazwy domenowej.
 * @param $resolve Boolean Wyszukać nazwy domenowe?
 */
function outputAddresses($resolve) {
    // ...
}
```

To całkiem niezłe rozwiązanie, pod warunkiem że programiści mający używać klasy są uważnymi czytelnikami dokumentacji.

Wreszcie można też zmodyfikować metodę `outputAddreses()` tak, by rygorystycznie traktowała typ danych, dostarczony za pośrednictwem argumentu `$resolve`. W starszych niż 7 wersjach PHP w przypadku typów elementarnych, takich jak wartości boolowskie, dało się to zrobić tylko w jeden sposób. Trzeba było napisać kod weryfikujący przekazane dane i podejmujący jakieś działania, gdy typ tych danych różnił się od oczekiwanego:

```
function outputAddresses($resolve)
{
    if (!is_bool($resolve)) {
    // podejmij drastyczne działania
    }
    // ...
}
```

Takie podejście można zastosować w celu wymuszenia na kodzie klienckim dostarczenia w argumencie `$resolve` właściwego typu danych, a gdy tak się nie stanie — wyświetlenia ostrzeżenia.

■ **Uwaga** W następnej części rozdziału, „Typy obiektowe", opiszę znacznie lepszy sposób ograniczania typów argumentów przekazywanych do metod i funkcji.

Wyręczenie wywołującego i konwersja argumentu typu łańcuchowego na typ logiczny łagodniej traktuje użytkowników, ale prowokuje szereg kolejnych problemów. Udostępniając mechanizm konwersji, skazujemy się na odgadywanie intencji wywołującego. Narzucając mu stosowanie typu logicznego, dajemy mu z kolei wolną rękę co do sposobu odwzorowywania wartości logicznych w ciągach znaków — klient sam decyduje, czy to dopuszczalne i jakie słowo reprezentuje dla niego „prawdę". Metoda `outputAddresses()` może zaś skupić się na swym podstawowym zadaniu, do którego została powołana. Tego rodzaju skupienie na własnych zadaniach z celowym ignorowaniem szerszego kontekstu jest ważną zasadą programowania obiektowego i będę się na nią często w książce powoływać.

W istocie zaś strategie obsługi typów argumentów powinny być z jednej strony uzależnione od ważności ewentualnych błędów, a z drugiej — od korzyści związanych z elastycznością kodu. PHP potrafi rzutować wartości pomiędzy większością elementarnych typów, zależnie od zastanego kontekstu wykorzystania wartości. Na przykład liczby w ciągach znaków, jeśli ciągi te występują w wyrażeniach arytmetycznych, są konwertowane na postać ich całkowitych i zmiennoprzecinkowych odpowiedników. W kodzie można polegać na tej konwersji, czyniąc go odpornym na szereg błędów typowania. Jeśli jednak któryś z argumentów metody ma być tablicą, nie można się spodziewać, że PHP dokona konwersji dowolnej wartości do sensownej tablicy — tego rodzaju pobłażliwość w dostosowaniu typów może prowadzić do istnej powodzi błędów w ciele metody.

Trzeba więc wyznaczyć pewien punkt równowagi pomiędzy pobłażliwością względem niesfornych wywołujących a bezwzględnym wymuszaniem odpowiedniego typu. Bardzo istotną rolę gra tutaj dokumentacja, która nie powinna pozostawiać u wywołującego wątpliwości co do pożądanego typu argumentu wywołania metody.

Niezależnie od sposobu radzenia sobie z tego rodzaju problemami trzeba mieć świadomość, że mimo bardzo luźnej kontroli typów w PHP typ ma istotne znaczenie. Ba, fakt liberalnego traktowania typów w PHP jeszcze to znaczenie potęguje. Nie można przy tym w zadaniu wykrywania błędów typowania zdawać się na kompilator. To programista musi oszacować ewentualny wpływ niedopasowania typów na wykonanie metody i odpowiednio do szacunków dobrać metodę obsługi typów argumentów. Nie sposób przy tym wymagać od wywołujących przenikliwości właściwej telepatom, stąd konieczność przygotowania kodu na okoliczność niepożądanych typów argumentów.

Typy obiektowe

Jako że argument wywołania funkcji może reprezentować wartość dowolnego typu elementarnego, może też domyślnie reprezentować obiekt dowolnego typu. Taka elastyczność ma swoje zalety, ale powoduje też problemy, zwłaszcza w kontekście definicji metody.

Wyobraźmy sobie metodę pewnej klasy pomocniczej, przeznaczonej do manipulowania obiektami klasy `ShopProduct`:

```
// listing 03.18
class ShopProductWriter
{
    public function write($shopProduct)
    {
        $str  = $shopProduct->title . ": "
            . $shopProduct->getProducer()
            . " (" . $shopProduct->price . ")\n";
        print $str;
    }
}
```

Klasę tę możemy przetestować kodem:

```
// listing 03.19
$product1 = new ShopProduct("Moja Antonia", "Willa", "Cather", 5.99);
$writer = new ShopProductWriter();
$writer->write($product1);
```

Otrzymamy:

```
Moja Antonia: Willa Cather (5.99)
```

Klasa `ShopProductWriter` zawiera tylko jedną metodę — `write()`. Metoda ta przyjmuje za pośrednictwem argumentu wywołania obiekt klasy `ShopProduct`, a odwołując się do jego składowych i metod, konstruuje ciąg podsumowujący wartość obiektu. Nazwa parametru metody, `$shopProduct`, sygnalizuje co prawda spodziewany typ obiektu, ale w żaden sposób go nie wymusza. Oznacza to, że argumentem wywołania metody mógłby być dowolny typ prosty albo obiektowy, a jego faktyczny typ mógłby się objawić dopiero przy próbie użycia go w operacji zakładającej obecność obiektu klasy `ShopProduct`. Tyle że jeszcze przed użyciem argumentu metoda może wykonać pewne operacje na bazie założenia, że ma do czynienia z obiektem odpowiedniej klasy.

■ **Uwaga** Metodę `write()` można by dodać bezpośrednio do klasy `ShopProduct`. Nie zrobimy tego jednak ze względu na podział odpowiedzialności. Klasa `ShopProduct` ma realizować zadania zarządzania danymi produktów; za wypisywanie danych o produktach odpowiedzialna jest klasa `ShopProductWriter`. Znaczenie i przydatność wyraźnego podziału odpowiedzialności stanie się bardziej oczywiste po lekturze dalszej części rozdziału.

Problem niemożności wymuszenia typu w wywołaniu metody wyeliminowano w PHP5 wraz z mechanizmem deklarowania oczekiwanego typu klasy (wtedy nosił on nazwę *type hints*). Aby dodać deklarację typu do argumentu metody, należy po prostu poprzedzić argument, którego typ chcemy narzucić, nazwą odpowiedniej klasy. Metodę `write()` można by więc przepisać tak:

```
// listing 03.20
public function write(ShopProduct $shopProduct)
{
  // ...
}
```

Teraz metoda `write()` nie będzie akceptowała w roli argumentów wywołania obiektów klas innych niż `ShopProduct`. Możemy to sprawdzić, prowokując niepoprawne wywołanie:

```
// listing 03.21
class Wrong
{
}
$writer = new ShopProductWriter();
$writer->write(new Wrong());
```

Z racji obecności w ciele metody `write()` deklaracji typu przekazanie w wywołaniu obiektu nieodpowiedniej (`Wrong`) klasy sprowokuje krytyczny błąd programu:

```
TypeError: Argument 1 passed to ShopProductWriter::write() must be an instance of ShopProduct,
instance of Wrong given, called in Runner.php on...
```

Możemy teraz darować sobie testowanie typu argumentu przed przystąpieniem do jego przetwarzania. Otrzymujemy też bardziej przejrzystą dla użytkownika sygnaturę metody — użytkownik może na jej podstawie od razu wnioskować co do oczekiwanego w wywołaniu typu, bez konieczności uciekania się do dokumentacji. A ponieważ deklaracja typu jest rygorystycznie przestrzegana, unikamy nie zawsze łatwych do wykrycia błędów charakterystycznych dla błędów typowania.

Choć tak zautomatyzowana kontrola poprawności typów skutecznie eliminuje obszerną kategorię błędów, trzeba zdawać sobie sprawę, że deklaracje typów są kontrolowane w czasie wykonania programu. Oznacza to, że błąd naruszenia deklaracji zostanie wykryty i zgłoszony dopiero w momencie, w którym nastąpi wywołanie metody z obiektem nieodpowiedniej klasy. Jeśli przypadkiem niefortunne wywołanie `write()` będzie osadzone w klauzuli warunkowej uruchamianej jedynie w Boże Narodzenie, możesz spodziewać się pracowitych świąt — wcześniej błąd pozostanie najprawdopodobniej ukryty.

Uzbrojeni w możliwość deklarowania typów skalarnych możemy wprowadzić pewne ograniczenia w klasie `ShopProduct`.

```
// listing 03.22
class ShopProduct
{
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price = 0;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float $price
    ) {
        $this->title = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName = $mainName;
        $this->price = $price;
```

```
    }
    // ...
}
```

Przy takiej formie konstruktora możemy mieć pewność, że argumenty `$title`, `$firstName` i `$mainName` zawsze będą zawierały łańcuchy znaków, a argument `$price` będzie typu `float`. Można się o tym przekonać, podejmując próbę utworzenia obiektu `ShopProduct` na bazie błędnych danych:

```
// listing 03.23
// to się nie uda
$product = new ShopProduct("tytuł", "imię", "nazwisko", []);
```

Przy próbie utworzenia egzemplarza obiektu `ShopProduct` do konstruktora zostały przekazane trzy łańcuchy znaków. Polegliśmy jednak na ostatnim argumencie, przekazując zamiast liczby typu `float` pustą tablicę. Dzięki deklaracjom typów PHP nie pozwoli nam na takie postępowanie:

```
TypeError: Argument 4 passed to ShopProduct::__construct() must be of the type float, array given,
called in...
```

Domyślnie PHP będzie jawnie rzutował argumenty na wymagany typ tam, gdzie to możliwe. Jest to przykład pewnego konfliktu między bezpieczeństwem a elastycznością (o czym była mowa wcześniej). Nowa wersja klasy `ShopProduct` „po cichu" zamieni łańcuch znaków na liczbę typu `float`. Na przykład taka próba konkretyzacji obiektu zakończy się powodzeniem:

```
// listing 03.24
$product = new ShopProduct("tytuł", "imię", "nazwisko", "4.22");
```

Za kulisami łańcuch znaków `"4.22"` został zamieniony na wartość zmiennoprzecinkową `4.22`.

Jak dotąd jest nieźle. Wróćmy jednak do problemu, który napotkaliśmy w związku z klasą `AddressManager`. Łańcuch `"false"` został po cichu zamieniony na boolowską wartość `true`. Domyślnie takie coś nadal będzie miało miejsce, jeśli zastosujemy deklarację typu `bool` w metodzie `AddressManager::outputAddresses()` w następujący sposób:

```
// listing 03.25
    public function outputAddresses(bool $resolve)
    {
    // ...
    }
```

Weźmy teraz wywołanie, w którym przekażemy łańcuch znaków, na przykład tak:

```
// listing 03.26
$manager->outputAddresses("false");
```

Ze względu na automatyczne rzutowanie kod ten jest funkcjonalnie identyczny z tym, który przekazywał wartość boolowską `true`.

Deklaracje typów skalarnych można potraktować bardziej rygorystycznie, ale tylko w obrębie danego pliku. W poniższym przykładzie metoda `outputAddresses()` została wywołana ponownie z argumentem w postaci łańcucha znaków, ale po włączeniu rygorystycznej kontroli typów:

```
// listing 03.27
declare(strict_types=1);
$manager->outputAddresses("false");
```

Ze względu na rygorystyczną kontrolę typów spowoduje to wygenerowanie błędu `TypeError`:

```
TypeError: Argument 1 passed to AddressManager::outputAddresses() must be of the type boolean,
string given, called in...
```

■ **Uwaga** Deklaracja `strict_types` dotyczy pliku, z którego nastąpiło wywołanie, a nie pliku, w którym została zaimplementowana dana metoda albo funkcja. Obowiązek wymuszenia zgodności typów leży więc po stronie kodu klienckiego.

Można potraktować jakiś argument jako opcjonalny, ale zarazem narzucić mu konkretny typ (jeśli argument ten zostanie przekazany). Da się to zrobić dzięki podaniu wartości domyślnej argumentu:

```
// listing 03.28
class ConfReader
{
    public function getValues(array $default = null)
    {
        $values = [];
    // zrób coś, aby pobrać wartości
    // scal dostarczone wartości domyślne (zawsze będą one miały postać tablicy)
        $values = array_merge($default, $values);
        return $values;
    }
}
```

W tabeli 3.2 zostały wymienione deklaracje typów, obsługiwane przez PHP.

Tabela 3.2. *Deklaracje typów*

Deklaracja typu	Od wersji	Opis
`array`	5.1	Tablica. Dopuszcza wartość domyślną `null` albo tablicę.
`int`	7.0	Liczba całkowita. Dopuszcza wartość domyślną `null` albo liczbę całkowitą.
`float`	7.0	Liczba zmiennoprzecinkowa (z częścią ułamkową). Zaakceptowana zostanie jednak także liczba całkowita — nawet po włączeniu rygorystycznej kontroli typów. Dopuszcza wartość domyślną `null`, liczbę zmiennoprzecinkową albo liczbę całkowitą.
`callable`	5.4	Kod wywoływalny (taki jak funkcja anonimowa). Dopuszcza wartość domyślną `null`.
`bool`	7.0	Wartość boolowska. Dopuszcza wartość domyślną `null` albo wartość boolowską.
`string`	5.0	Dane znakowe. Dopuszcza wartość domyślną `null` albo łańcuch znaków.
`self`	5.0	Odwołanie do klasy macierzystej.
[*typ klasy*]	5.0	Typ klasy albo interfejsu. Dopuszcza wartość domyślną `null`.

Kiedy pisaliśmy o deklaracji typu klasy, traktowaliśmy typy i klasy jako pojęcia równoznaczne. Tymczasem pomiędzy typami a klasami istnieje zasadnicza różnica. Otóż definiując klasę, definiuje się równocześnie typ, ale typ jako taki może opisywać całą rodzinę klas. Mechanizm grupowania wielu klas w obrębie jednego typu nosi nazwę dziedziczenia. Będzie on tematem następnego podrozdziału.

Dziedziczenie

Dziedziczenie to mechanizm wyprowadzania jednej bądź wielu klas pochodnych z pewnej wspólnej klasy bazowej.

Klasa dziedzicząca po innej klasie staje się jej podklasą. Owa relacja często opisywana jest w oparciu o relację rodzic – dziecko. Owo „dziecko” (klasa potomna czy też pochodna) jest wyprowadzone z klasy „rodzica”

(klasy nadrzędnej albo bazowej) i dziedziczy jej składowe i metody. Klasa pochodna zazwyczaj uzupełnia elementy odziedziczone własnymi składowymi i metodami — mówi się wtedy o „rozszerzaniu" klasy bazowej[1].

Zanim zagłębimy się w składnię dziedziczenia, powinniśmy rozpoznać problemy, w których rozwiązywaniu dziedziczenie okazuje się pomocne.

Problemy związane z dziedziczeniem

Wróćmy do naszej klasy `ShopProduct`. Na razie jest ona dość ogólna, ponieważ nie ogranicza asortymentu produktów (mimo że dotychczas jej obiekty reprezentowały asortyment księgarski).

```
// listing 03.29
$product1 = new ShopProduct("Moja Antonia", "Willa", "Cather", 5.99);
$product2 = new ShopProduct("Exile on Coldharbour Lane", "The", "Alabama 3", 10.99);

print "Autor: " . $product1->getProducer() . "\n";
print "Wykonawca : " . $product2->getProducer() . "\n";
```

Program wypisze na wyjściu:

```
Autor     : Willa Cather
Wykonawca : The Alabama 3
```

Rozdzielenie nazwy autora („producenta") na dwie części sprawdza się dla książek i nawet dla albumów CD. Możemy dzięki niemu wyszukiwać i porządkować asortyment wg „Alabama 3" i „Cather", pozbywając się mniej znaczących „The" i „Willa". Wygoda to zazwyczaj znakomita strategia projektowa, nie musimy więc na razie przejmować się dostosowaniem projektu klasy `ShopProduct` do artykułów innych rodzajów.

Gdybyśmy jednak nasz przykład uzupełnili o pewne dodatkowe wymagania, rzecz szybko by się skomplikowała. Załóżmy na przykład, że obiekty klasy `ShopProducer` powinny jednak przechowywać dodatkowo informacje charakterystyczne dla ich asortymentu — inne w przypadku książek (np. liczba stron), inne w przypadku albumów CD (np. czas nagrania). Różnic może być znacznie więcej, ale i te wystarczą do ilustracji problemu.

W jaki sposób powinniśmy rozszerzyć klasę, aby dało się odzwierciedlić w niej nowe wymagania? Niemal natychmiast na myśl przychodzą dwie możliwości. Pierwsza polega na zebraniu w klasie `ShopProduct` wszelkich możliwych składowych. Druga zakłada podział klasy na dwie osobne.

Spróbujmy pierwszego sposobu — połączenia w jednej klasie składowych charakterystycznych dla płyt i książek:

```
// listing 03.30
class ShopProduct
{
    public $numPages;
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float $price,
        int $numPages = 0,
        int $playLength = 0
    ) {
        $this->title            = $title;
```

[1] Choć właściwsze byłoby mówienie o „specjalizacji" — *przyp. tłum.*

```
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
        $this->playLength        = $playLength;
    }
    public function getNumberOfPages()
    {
        return $this->numPages;
    }
    public function getPlayLength()
    {
        return $this->playLength;
    }
    public function getProducer()
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
}
```

W definicji klasy pojawiły się metody dające dostęp do składowych `$numPages` i `$playLength`; kod ten ilustruje też pewną nadmiarowość. Otóż obiekt konkretyzowany z takiej klasy będzie zawierał nadmiarową metodę, a w przypadku obiektów dla płyt CD konstruktor będzie przyjmował niepotrzebny argument: obiekt reprezentujący płytę CD będzie utrzymywać informacje i funkcje właściwe dla obiektów książek (tu: liczbę stron) i odwrotnie — obiekt reprezentujący w istocie książkę będzie niepotrzebnie przechowywał długość nagrania. Na razie zapewne możemy się z taką nadmiarowością pogodzić. Ale co, jeśli asortyment zostanie rozszerzony na kolejne kategorie produktów, a wraz z nimi pojawią się w klasie kolejne składowe i metody? Klasa nadmiernie się rozrośnie i stanie się po prostu niewygodna w użyciu.

Jak widać, scalanie w jednej klasie danych i funkcji różnych klas prowadzi do rozszerzenia obiektów o nadmiarowe i zbędne składowe i metody.

Problem nie kończy się jednak na nadmiarowości danych. Cierpi również funkcjonalność klasy. Weźmy choćby metodę zestawiającą informacje o produkcie. Niech dział sprzedaży zażyczy sobie możliwości generowania podsumowania informacji o artykule na potrzeby wystawianych w dziale faktur. W opisie albumu CD ma znaleźć się długość nagrania, a w opisie książki — liczba stron. Trzeba więc będzie przewidzieć różne implementacje zestawień dla każdego rodzaju asortymentu. Można by spróbować wygospodarować w klasie znacznik informujący o formacie obiektu, jak w tym przykładzie:

```
// listing 03.31
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;
}
```

Alternatywnie, aby poprawnie ustawić składową `$type`, moglibyśmy sprawdzić wartość argumentu wywołania konstruktora dla parametru `$numPages`. Słowem, dalej niepotrzebnie „rozdymamy" i komplikujemy klasę `ShopProduct`, a w miarę dokładania różnych formatów obiektów różnice funkcjonalne pomiędzy nimi będą coraz trudniejsze do ujęcia w spójnej implementacji. Może więc lepiej byłoby spróbować innego sposobu?

Ponieważ klasa `ShopProduct` zaczyna przypominać siłowe sklejenie dwóch klas, możemy spróbować podzielić ją na dwoje. Moglibyśmy podejść do zadania tak:

```
// listing 03.32
class CdProduct
{
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float  $price,
        int      $playLength
    ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->playLength        = $playLength;
    }
    public function getPlayLength()
    {
        return $this->playLength;
    }
    public function getSummaryLine()
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": czas nagrania - {$this->playLength}";
        return $base;
    }
    public function getProducer()
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
}
```

```
// listing 03.33
class BookProduct
{
    public $numPages;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float   $price,
        int      $numPages
    ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
```

```
    }
    public function getNumberOfPages()
    {
        return $this->numPages;
    }
    public function getSummaryLine()
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": liczba stron - {$this->numPages}";
        return $base;
    }
    public function getProducer()
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
}
```

Rozwiązaliśmy problem rosnącej złożoności klasy, ale zapłaciliśmy za to pewną cenę. Teraz możemy tworzyć osobne wersje metody `getSummaryLine()` bez potrzeby kontrolowania w jej ciele znacznika właściwego formatu obiektu. Klasa nie utrzymuje też zbędnych składowych i metod.

Wspomnianą ceną jest powielenie kodu. Metoda `getProducer()` w obu klasach ma identyczny kod. Każdy z konstruktorów ustawia w identyczny sposób pewien podzbiór składowych obiektu. To istotna wada kodu i warto się jej pozbyć.

Skoro metoda `getProducer()` zachowuje się identycznie dla każdej z klas, to jakakolwiek zmiana tego zachowania będzie musiała być zaimplementowana z osobna we wszystkich tych klasach. Prędzej czy później podczas tej synchronizacji będziemy przeklinać podjętą decyzję.

Jeśli zaś jesteś przekonany, że poradzisz sobie z duplikacją kodu, to nie możesz zapomnieć, że teraz zamiast jednego typu mamy dwa różne (mimo podobieństw) typy.

Wróćmy do klasy `ShopProductWriter`. Jej metoda `write()` została przystosowana do pracy na obiektach pojedynczego typu — obiektach klasy `ShopProduct`. W jaki sposób zmusić ją do obsługi obiektów dwóch różnych klas? Możemy oczywiście usunąć z definicji metody deklarację typu argumentu, ale wtedy będziemy musieli w pełni zaufać wywołującemu — program będzie poprawny jedynie wtedy, kiedy do metody będą przekazywane obiekty właściwych typów. Możemy dokonywać kontroli typów w ciele metody:

```
// listing 03.34
class ShopProductWriter
{
    public function write($shopProduct)
    {
        if (
            ! ($shopProduct instanceof CdProduct)  &&
            ! ($shopProduct instanceof BookProduct)
        ) {
            die("Przekazano niewłaściwy typ danych");
        }
        $str  = "{$shopProduct->title}: "
            . $shopProduct->getProducer()
            . " ({$shopProduct->price})\n";
        print $str;
    }
}
```

W przykładzie wykorzystaliśmy operator `instanceof`. Wywołanie tego operatora daje wartość `true`, jeśli jego lewy operand jest egzemplarzem klasy występującej w roli prawego operandu.

Raz jeszcze zostaliśmy więc zmuszeni do wprowadzenia dodatkowego poziomu złożoności w kodzie. Nie tylko musimy testować przynależność przekazanego w wywołaniu `write()` obiektu do jednego z dwóch

typów, ale i ufać, że żaden z tych typów nie zaniecha obsługi wykorzystywanych przez nas składowych i metod. Rzecz wyglądała znacznie lepiej, kiedy żądaliśmy przekazania w wywołaniu konkretnego typu, większą mieliśmy też pewność, że typ ten — klasa `ShopProduct` — posiada taki, a nie inny interfejs.

Odmiany książkowa i płytowa klasy `ShopProduct` nie współgrają ze sobą, ale mimo wszystko zdaje się, że mogą ze sobą przynajmniej koegzystować. Lepiej byłoby jednak, gdybyśmy i obiekty reprezentujące książki, i obiekty płyt muzycznych mogli traktować jak egzemplarze jednej klasy, ale wyposażone w nieco odmienną implementację stosowną do formatu wymaganego w prezentacji asortymentu towarów. Chcielibyśmy więc móc zdefiniować wspólny zestaw funkcji i cech, unikając duplikacji kodu, ale równocześnie umożliwić rozgałęzienie implementacji niektórych wywołań metod zależnie od formatu obiektu. Rozwiązaniem jest dziedziczenie.

Stosowanie dziedziczenia

Pierwszym etapem konstrukcji hierarchii dziedziczenia jest identyfikacja tych elementów klasy bazowej, które nie są na tyle uniwersalne, aby dały się identycznie obsługiwać we wszystkich egzemplarzach.

W naszej powstałej swego czasu klasie `ShopProducer` mieliśmy, na przykład, kolidujące ze sobą metody `getPlayLength()` i `getNumberOfPages()`. Pamiętamy też, że wspólna dla wszystkich obiektów metoda `getSummaryLine()` wymagała różnych implementacji dla różnych formatów obiektów. Te trzy różnice mogą nam posłużyć do wyodrębnienia klasy bazowej i dwóch klas pochodnych:

```
// listing 03.35
class ShopProduct
{
    public $numPages;
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float   $price,
        int      $numPages = 0,
        int      $playLength = 0
    ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
        $this->playLength        = $playLength;
    }
    public function getProducer()
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
    public function getSummaryLine()
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}
```

```
// listing 03.36
class CdProduct extends ShopProduct
{
    public function getPlayLength()
    {
        return $this->playLength;
    }
    public function getSummaryLine()
    {
        $base  = "{$this->title} ( {$this->producerMainName},
        $base .= "{$this->producerFirstName} )";
        $base .= ": czas nagrania - {$this->playLength}";
        return $base;
    }
}
```

```
// listing 03.37
class BookProduct extends ShopProduct
{
    public function getNumberOfPages()
    {
        return $this->numPages;
    }
    public function getSummaryLine()
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": liczba stron - {$this->numPages}";
        return $base;
    }
}
```

Utworzenie klasy pochodnej wymaga opatrzenia deklaracji klasy słowem `extends`. W powyższym przykładzie utworzyliśmy w ten sposób dwie nowe klasy: `BookProduct` i `CdProduct`. Obie rozszerzają i uzupełniają klasę `ShopProduct`.

Ponieważ w klasach pochodnych zabrakło definicji konstruktorów, w momencie konkretyzacji obiektów tych klas wywoływany jest automatycznie konstruktor klasy bazowej. Klasy pochodne dziedziczą bowiem dostęp do wszystkich publicznych i chronionych metod klasy bazowej (z wyjątkiem składowych i metod prywatnych). Oznacza to, że możemy wywoływać metodę `getProducer()` na rzecz obiektu konkretyzowanego z klasy `CdProduct`, choć sama metoda `getProducer()` jest zdefiniowana nie w `CdProduct`, a w `ShopProduct`.

```
// listing 03.38
$product2 = new CdProduct(
    "Exile on Coldharbour Lane",
    "The",
    "Alabama 3",
    10.99,
    0,
    60.33
);
print "Wykonawca: {$product2->getProducer()}\n";
```

Jak widać, obie klasy pochodne dziedziczą zachowanie po rodzicu. Obiekt klasy `BookProduct` możemy więc traktować jak wcielenie obiektu klasy `ShopProduct`. I dlatego też możemy przekazywać obiekty klasy `BookProduct` bądź `CdProduct` w wywołaniu metody `write()` klasy `ShopProductWriter`.

Zauważmy, że w klasach `CdProduct` i `BookProduct` nastąpiło przesłonięcie metody `getSummaryLine()` jej implementacjami odpowiednimi dla tych klas. Sęk w tym, że klasy pochodne mogą nie tylko rozszerzać i uzupełniać, ale i modyfikować zachowanie klas nadrzędnych.

Implementacja tej metody w klasie bazowej wydaje się nadmiarowa, skoro i tak jest przepisywana w obu klasach pochodnych. Niemniej jednak ta bazowa implementacja udostępnia najbardziej podstawową realizację danej funkcji, dostępną do użycia w klasach pochodnych. Obecność metody w klasie bazowej daje też gwarancję, że wszelkie obiekty klasy `ShopProduct` (i klas pochodnych) w kodzie klienckim będą posiadać metodę `getSummaryLine()`. Później przekonamy się, że taką gwarancję można wymusić bez implementowania metody w klasie bazowej. Każda klasa pochodna `ShopProduct` dziedziczy komplet składowych klasy „rodzica". Dlatego zarówno klasa `CdProduct`, jak i `BookProduct` mogą w swoich implementacjach metody `getSummaryLine()` odwoływać się do składowej `$title`.

Dziedziczenie może z początku być koncepcją niejasną. Definiując klasę rozszerzającą inną klasę, gwarantujemy, że obiekt tejże nowej klasy będzie w pierwszym rzędzie określany cechami definiowanymi w klasie pochodnej, a dopiero w drugiej kolejności tymi z klasy bazowej. Można też zastosować inną analogię — gdybyśmy chcieli samodzielnie rozprowadzić wywołanie `$product2->getProducer()`, nie znaleźlibyśmy takiej metody w klasie `CdProduct`, więc wywołanie przenieślibyśmy do „domyślnej" implementacji tej metody, zdefiniowanej w `ShopProduct`. Ale już wywołanie `$product2->getSummaryLine()` możemy zrealizować za pomocą metody z klasy `CdProduct`.

To samo dotyczy odwołań do składowych. Występującego w metodzie `getSummaryLine()` klasy `BookProduct` odwołania do składowej `$title` nie można zrealizować w ramach klasy `BookProduct`; jest ona pobierana z klasy bazowej. Pozostawienie jej w klasie bazowej ma uzasadnienie, ponieważ inaczej trzeba by ją dublować we wszystkich pochodnych.

Rzut oka na konstruktor klasy bazowej ujawnia jednak, że wciąż w klasie bazowej obsługujemy dane, których obsługa powinna zostać przeniesiona do klas pochodnych. Otóż klasa `BookProduct` powinna przejąć obsługę argumentu i składowej `$numPages`, a składowa `$playLength` powinna zostać wyodrębniona do klasy `CdProduct`. W tym celu trzeba by w klasach pochodnych zdefiniować ich własne konstruktory.

Dziedziczenie a konstruktory

Definiując konstruktor klasy pochodnej, trzeba wziąć na siebie odpowiedzialność za przekazanie argumentów do wywołania konstruktora klasy bazowej. Jeśli to zaniedbamy, otrzymamy częściowo tylko skonstruowany obiekt.

Aby wywołać z wnętrza klasy pochodnej metodę klasy bazowej, musimy najpierw poznać sposób odwołania się do klasy jako takiej. W języku PHP służy do tego słowo kluczowe `parent`.

Aby odwołać się do metody w kontekście klasy, a nie obiektu, powinniśmy zamiast operatora `->` zastosować operator `::`.

```
parent::__construct()
```

Powyższy zapis oznacza więc: „wywołanie metody `__construct()` klasy bazowej". Spróbujmy zatem zmodyfikować nasz przykład tak, aby każda klasa odpowiadała jedynie za swoje własne składowe:

```
// listing 03.39
class ShopProduct
{
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    function __construct(
        $title,
        $firstName,
        $mainName,
        $price
    ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName
        $this->producerMainName  = $mainName;
        $this->price             = $price;
    }
    function getProducer()
```

```
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
    function getSummaryLine()
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}
```

// listing 03.40

```
class BookProduct extends ShopProduct
{
    public $numPages;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float   $price,
        int      $numPages
    ) {
        parent::__construct(
            $title,
            $firstName,
            $mainName,
            $price
        );
        $this->numPages = $numPages;
    }
    public function getNumberOfPages()
    {
        return $this->numPages;
    }
    public function getSummaryLine()
    {
        $base  = "{$this->title} ( $this->producerMainName, ";
        $base .= "$this->producerFirstName )";
        $base .= ": liczba stron – {$this->numPages}";
        return $base;
    }
}
```

// listing 03.41

```
class CdProduct extends ShopProduct
{
    public $playLength;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float   $price,
        int      $playLength
    ) {
        parent::__construct(
            $title,
            $firstName,
```

```
            $mainName,
            $price
        );
        $this->playLength = $playLength;
    }
    public function getPlayLength()
    {
        return $this->playLength;
    }
    public function getSummaryLine()
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": czas nagrania - {$this->playLength}";
        return $base;
    }
}
```

Każda klasa pochodna wywołuje w swoim konstruktorze konstruktor klasy bazowej, a dopiero potem przystępuje do ustawiania własnych składowych. Klasa bazowa troszczy się wyłącznie o swoje dane. Klasy pochodne są zaś w ogólności specjalizacjami klas bazowych. Należy więc unikać ujmowania w klasach bazowych (jako ogólniejszych) specjalistycznej wiedzy o klasach pochodnych.

■ **Uwaga** W wersjach poprzedzających PHP5 konstruktory miały nazwy zgodne z nazwami klas. Teraz nazwy metod konstrukcji zostały ujednolicone — konstruktor każdej klasy nazywa się `__construct()`. Gdyby zechcieć skorzystać z przestarzałej składni, wywołanie konstruktora klasy bazowej wiązałoby kod klasy pochodnej z tą konkretną klasą: `parent::ShopProduct();`. W PHP7 poprzednia składnia konstruktorów została uznana za przestarzałą i nie powinna być używana.

Wywołania metod przesłoniętych

Słowo kluczowe `parent` można stosować w odwołaniach do wszelkich metod klasy bazowej, które zostały przesłonięte w klasie pochodnej. Niekiedy bowiem zamiast całkiem przesłaniać metodę klasy bazowej, chcemy jedynie uzupełnić jej działanie. Możemy się wtedy we własnej implementacji wesprzeć wersją metody z klasy bazowej. Mimo ulepszeń wprowadzonych do naszej hierarchii klas wciąż mamy do czynienia z pewną duplikacją kodu — dochodzi do niej w ramach metody `getSummaryLine()`. Tymczasem zamiast powtarzać kod w klasach pochodnych, moglibyśmy odwołać się do kodu klasy `ShopProduct` i tylko uzupełnić go stosownie do potrzeb klasy pochodnej:

```
// listing 03.42
// Klasa ShopProduct...
    function getSummaryLine()
    {
        $base = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
```

```
// listing 03.43
// Klasa BookProduct...
    public function getSummaryLine()
   {
        $base = parent::getSummaryLine();
        $base .= ": liczba stron - {$this->numPages}";
        return $base;
    }
```

Podstawowe zadania metody `getSummaryLine()` zdefiniowaliśmy w klasie `ShopProduct`. Teraz zamiast powtarzać jej kod w klasach `CdProduct` i `BookProduct`, możemy po prostu wywołać wersję z klasy bazowej i uzupełnić otrzymany z takiego wywołania ciąg danymi charakterystycznymi dla klasy pochodnej.

Znając już podstawowe zasady dziedziczenia, możemy wrócić do zagadnienia widoczności metod i składowych.

Zarządzanie dostępem do klasy — słowa public, private i protected

Jak dotąd wszystkie składowe i metody klas deklarowaliśmy jako publiczne (ze słowem kluczowym `public`). Dostęp publiczny do klasy jest zakładany domyślnie — jako publiczne są też traktowane te składowe, które są deklarowane z wykorzystaniem obowiązującego w PHP4 słowa `var`.

Tymczasem elementy klas mogą być deklarowane jako publiczne (`public`), ale również jako chronione (`protected`) i prywatne (`private`):

- Metody i składowe publiczne są dostępne niezależnie od kontekstu.
- Metody i składowe prywatne są dostępne jedynie z wnętrza zawierającej je klasy. Dostęp do nich jest odmawiany nawet klasom pochodnym.
- Metody i składowe chronione są dostępne z klasy, w której są deklarowane, oraz z jej klas pochodnych. Dostępu nie uzyskuje jednak kod zewnętrzny wobec danej klasy.

Czy różnicowanie widoczności może być w ogóle przydatne? Owszem, ponieważ odpowiednio stosowane słowa regulujące widoczność pozwalają na eksponowanie z klasy jedynie tych jej elementów, które są potrzebne użytkownikom obiektów klasy i jako takie stanowią jej interfejs.

Uniemożliwiając użytkownikom zewnętrznym odwoływanie się do niektórych składowych, możemy zapobiegać błędom. Wyobraźmy sobie, że obiekty `ShopProduct` miałyby przechowywać informacje o rabatach. Informacje te miałyby być przechowywane w składowych `$discount`, których ustawienie następowałoby przez wywołanie metody `setDiscount()`:

```
// listing 03.44
// Klasa ShopProduct
    public $discount = 0;
// ...
    public function setDiscount(int $num)
    {
        $this->discount = $num;
    }
```

Uzbrojeni w mechanizm przyznawania rabatów możemy zdefiniować metodę `getPrice()`, która wyceni artykuł z uwzględnieniem owego rabatu:

```
    public function getPrice()
    {
        return ($this->price - $this->discount);
    }
```

Pojawia się jednak problem. Otóż użytkownicy zewnętrzni powinni widzieć jedynie ostateczne ceny (uwzględniające rabaty), tymczasem użytkownik mający dostęp do obiektu może — zamiast wywoływać metodę `getPrice()` — pójść na skróty i odwołać się bezpośrednio do składowej przechowującej cenę:

```
print "Cena artykułu wynosi {$product1->price}\n";
```

Co spowoduje wyprowadzenie mylącej (bo nieuwzględniającej rabatów) ceny. Takim przypadkom możemy zapobiec, deklarując `$price` jako składową prywatną i uniemożliwiając dostęp do niej z zewnątrz. Zmusi to użytkowników do korzystania z metody `getPrice()`. Próba odwołania się do prywatnej składowej spoza klasy `ShopProduct` będzie bowiem nieskuteczna. Dla świata zewnętrznego składowa ta przestanie po prostu istnieć.

Ukrywanie składowych jako prywatnych może jednak okazać się nadgorliwością. Do składowych prywatnych nie mają dostępu nawet klasy pochodne. Wyobraźmy sobie, że zgodnie z założeniami biznesowymi książki

zostały wyłączone z wszelkich promocji. Pomysł taki moglibyśmy zrealizować, przesłaniając w klasie `BookProduct` metodę `getPrice()` z pominięciem rabatu. Metoda ta musi jednak mieć dostęp do składowej `$price`:

```
// listing 03.45
// BookProduct
    public function getPrice()
    {
        return $this->price;
    }
```

Jeśli składowa `$price` zostałaby zadeklarowana jako prywatna w klasie `ShopProduct`, to niestety powyższy kod byłby niepoprawny, gdyż dostęp do składowych prywatnych jest zablokowany nawet dla klas pochodnych. Rozwiązaniem byłoby zadeklarowanie `$price` jako składowej chronionej i jako takiej dostępnej z poziomu klas pochodnych. Trzeba przy tym pamiętać, że tak oznaczona składowa nie będzie w ogóle dostępna dla kodu spoza hierarchii dziedziczenia, w tym dla innych klas, które nie uczestniczą w tej hierarchii. Dostępność składowych i metod chronionych jest ograniczona do klasy, w której je zadeklarowano, i jej klas pochodnych.

Warto przyjąć regułę faworyzowania prywatności składowych i metod. To, co nie jest zabronione, jest dozwolone, lepiej więc domyślnie zaostrzać kryteria dostępu do składowych i rozluźniać je w miarę potrzeb. Wiele (zwykle większość) metod konstruowanych przez nas klas będzie metodami publicznymi, ale jeśli potrzeba ich udostępniania jest wątpliwa, lepiej z tego zrezygnować. Metoda udostępniająca lokalne funkcje danej klasy pozostałym metodom tej klasy nie powinna być widoczna dla użytkowników zewnętrznych — niech więc będzie albo całkiem prywatna, albo przynajmniej chroniona przed dostępem z zewnątrz.

Metody — akcesory

Jeśli nawet użytkownicy zewnętrzni muszą odwoływać się do wartości przechowywanych w obiektach klasy, nie znaczy to, że mają otrzymać pełny dostęp do tych składowych — niejednokrotnie lepiej regulować ten dostęp, definiując metody — akcesory — pośredniczące w odwołaniach do owych składowych.

Mieliśmy już okazję przekonać się o zaletach takich metod. Akcesor może bowiem nie tylko wprost udostępniać wartości, ale również filtrować je w zależności od okoliczności. Przykład takiego filtrowania mieliśmy w metodzie `getPrice()`.

Metody, o których mowa, mogą także służyć do wymuszania typu składowej. Deklaracje typów pozwalają narzucić typ argumentów w wywołaniach metod, ale składowa klasy może zawierać dane dowolnego typu. Jak pamiętamy, klasa `ShopProductWriter` wykorzystywała do wyprowadzania danych obiekt `ShopProduct`. Spróbujmy przerobić ją tak, aby mogła służyć do wyprowadzania wartości wielu obiektów `ShopProduct`:

```
// listing 03.46
class ShopProductWriter
{
    public $products = [];
    public function addProduct(ShopProduct $shopProduct)
    {
        $this->products[] = $shopProduct;
    }
    public function write()
    {
        $str =  "";
        foreach ($this->products as $shopProduct) {
            $str .= "{$shopProduct->title}: ";
            $str .= $shopProduct->getProducer();
            $str .= " ({$shopProduct->getPrice()})\n";
        }
        print $str;
    }
}
```

Klasa `ShopProductWriter` jest teraz znacznie bardziej użyteczna. Może przechowywać w swoich obiektach wiele egzemplarzy obiektów klasy `ShopProduct` i wypisywać dane wszystkich w jednym podejściu. Musimy jednak ufać, że użytkownicy kodu będą respektować nasze intencje; a pomimo że udostępniliśmy im metodę `addProduct()`, nie muszą wcale z niej korzystać — nie możemy zabronić programistom stosującym klasę `ShopProductWriter` bezpośredniego manipulowania składową `$products`. I narażamy się tym samym nie tylko na ryzyko wprowadzenia do składowej `$products` obiektów niepoprawnego typu, ale i zamazania całej tablicy albo zastąpienia jej wartością skalarną. Wszystkie te zagrożenia eliminujemy, oznaczając składową `$products` jako prywatną:

```
// listing 03.47
class ShopProductWriter {
    private $products = [];
// ...
```

Teraz nie ma możliwości zamazania składowej `$products` spoza klasy. Wszelkie odwołania do składowej muszą być realizowane za pośrednictwem metody `addProduct()`, która dzięki deklaracji typu klasy gwarantuje, że do składowej tej mogą zostać dodane tylko obiekty typu `ShopProduct`.

Klasy hierarchii ShopProduct

Zamknijmy rozdział wersjami deklaracji klas hierarchii `ShopProduct` uzupełnionymi o odpowiednie zabezpieczenia widoczności składowych i metod:

```
// listing 03.48
class ShopProduct
{
    private   $title;
    private   $producerMainName;
    private   $producerFirstName;
    protected $price;
    private   $discount = 0;

    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float  $price
    ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
    }
    public function getProducerFirstName()
    {
        return $this->producerFirstName;
    }
    public function getProducerMainName()
    {
        return $this->producerMainName;
    }
    public function setDiscount($num)
    {
        $this->discount = $num;
    }
    public function getDiscount()
    {
        return $this->discount;
    }
```

```
    public function getTitle()
    {
        return $this->title;
    }
    public function getPrice()
    {
        return ($this->price - $this->discount);
    }
    public function getProducer()
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
    public function getSummaryLine()
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}
```

// listing 03.49

```
class CdProduct extends ShopProduct
{
    private $playLength;
    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float  $price,
        int    $playLength
    ) {
        parent::__construct(
            $title,
            $firstName,
            $mainName,
            $price
        );
        $this->playLength = $playLength;
    }
    public function getPlayLength()
    {
        return $this->playLength;
    }
    public function getSummaryLine()
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": czas nagrania - {$this->playLength}";
        return $base;
    }
}
```

// listing 03.50

```
class BookProduct extends ShopProduct
{
    private $numPages;
    public function __construct(
        string $title,
```

```
        string $firstName,
        string $mainName,
        float  $price,
        int    $numPages
    ) {
        parent::__construct(
            $title,
            $firstName,
            $mainName,
            $price
        );
        $this->numPages = $numPages;
    }
    public function getNumberOfPages()
    {
        return $this->numPages;
    }
    public function getSummaryLine()
    {
        $base  = parent::getSummaryLine();
        $base .= ": liczba stron - $this->numPages";
        return $base;
    }
    public function getPrice()
    {
        return $this->price;
    }
}
```

W nowej wersji rodziny klas `ShopProduct` nie ma szczególnych nowinek. Wszystkie składowe są albo prywatne, albo chronione, co wymusiło uzupełnienie klas o odpowiednie akcesory składowych niepublicznych.

Podsumowanie

Niniejszy rozdział zawiera solidną dawkę podstaw, bo zaczynając od zupełnie pustej klasy, doszliśmy do rozbudowanej hierarchii dziedziczenia. Udało się wyjaśnić szereg istotnych kwestii, w tym tych związanych z typami i dziedziczeniem. Udało się też zilustrować elementy obsługi obiektów w PHP. A niebawem poznasz kolejne obiektowe mechanizmy PHP.

Skorowidz

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Ś

T

U

V

W

Z

Ż

Jeszcze kilka lat temu programowanie zorientowane obiektowo w PHP wydawało się dość karkołomnym zagadnieniem. Upowszechnienie licznych frameworków sprawiło jednak, że coraz więcej programistów tworzy aplikacje WWW, pisząc kod obiektowy w PHP. Frameworki są chętnie stosowane, gdyż wydaje się, że dzięki nim nie trzeba już poświęcać czasu na rozgryzanie szczegółów działania kodu. Niestety, bardzo często okazuje się, że bez zrozumienia zasad działania frameworków i bez umiejętności oceny jakości zastosowanych w nich rozwiązań projektant nie jest w stanie napisać poprawnie funkcjonującej aplikacji.

Trzymasz w ręku książkę przeznaczoną dla programistów, którzy chcą dogłębnie zrozumieć mechanizmy programowania obiektowego w języku PHP i dzięki temu tworzyć, testować oraz publikować efektywny kod. Dużo uwagi poświęcono tu wzorcom projektowym i ich stosowaniu. Opisano zalety wzorców i przedstawiono sporo klasycznych już rozwiązań. Nie zabrakło również omówienia narzędzi oraz metod postępowania, dzięki którym doskonały kod staje się udanym projektem. To wszystko sprawia, że niniejsza książka jest nieodzownym podręcznikiem dla każdego projektanta, który chce tworzyć niezawodne, eleganckie i efektywne aplikacje.

Najważniejsze zagadnienia:

- praca z obiektami: podstawy i zagadnienia zaawansowane
- wzorce projektowe, w tym korporacyjne i bazodanowe
- dobre i złe praktyki programistyczne
- zarządzanie wersjami i publikowanie kodu
- strategie testów automatycznych i ciągłej integracji

Matt Zandstra — jest programistą WWW, konsultantem technicznym i autorem książek. Jakiś czas temu był starszym programistą w Yahoo!, obecnie jest niezależnym konsultantem. Napisał kilka bardzo dobrze przyjętych książek o programowaniu w PHP. Specjalizuje się w rozwijaniu oprogramowania dla biznesu, zwłaszcza w PHP, Perlu i Javie, zarządzaniu treścią i programowaniu zorientowanym obiektowo. Chętnie udziela konsultacji dotyczących stosowania najlepszych praktyk programistycznych. Wraz z żoną Louise i dwojgiem dzieci mieszka w Liverpoolu.

Twój klucz do sukcesu:
eleganckie wzorce projektowe
i najlepsze praktyki programistyczne!

Helion
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

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

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

ISBN 978-83-283-3553-0
9 788328 335530

Informatyka w najlepszym wydaniu
cena: 89,00 zł