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

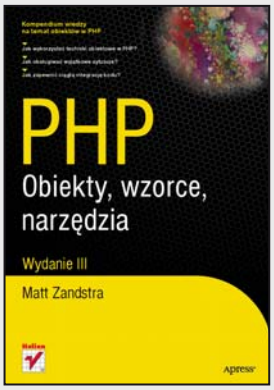
Wydanie III

Matt Zandstra

Helion

Apress

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 obiektową 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!

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

Spis treści

ROZDZIAŁ 6

Obiekty a projektowanie obiektowe

Znamy już dość szczegółowo mechanizmy obsługi obiektów w języku PHP, wypadałoby 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? Jakie 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 powiązaniami.

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 decydencie 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 obiektowością — 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. Załóżmy, 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 szczegóły implementacyjne.

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

```
klucz:wartość
```

Wystarczą 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.

Zdefiniujmy 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 `write()` 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 `writeParams()` 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 szerzej 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 pseudoobiektowym 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. Obiektowe systemy często wykorzystują programowe reprezentacje obiektów świata materialnego — w postaci klas `Person` (osoba), `Invoice` (faktura) czy `Shop` (sklep). Sugeruje to, że wyznaczanie 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 klas? 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 powołaliśmy 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 strategiom 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.

Niemożność określenia typów zwracanych oznacza, że wymienność 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 ujednolicaniu 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 ujednolicenia 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 strategie 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 nazewniczej — symptomem zamierzonej prywatności składowej było na przykład rozpoczynanie jej nazwy od znaku podkreślenia:

```
var $_niedotykac;
```

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 rozwiązania. Zaczyna się wybór funkcji przydatnych w implementacji rozwiązania, przypominanie sobie co sprytniejszych wyrażeń regularnych i poszukiwanie w repozytorium PEAR 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 powinny zaprzątać 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 zaburzania 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 wątpienia wpłynęło to na kształt interfejsu. I dobrze, bo nie sposób definiować interfejsów w całkowitym oderwaniu od wiadomych sobie aspektów systemu — doprowadziłoby to najprawdopodobniej do zbytniej ich generalizacji.

Sławna Banda Czworga (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 ścisłe 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? Skończy 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 powinieneś 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.

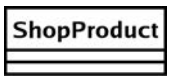
Przydatność UML-a przejawia się głównie w opisach struktur i wzorców, a to za sprawą diagramów klas. Uzyskiwana w tych diagramach przejrzystość intencji projektowych i podziału zadań rzadko daje się równie łatwo wyrazić w przykładowym kodzie.

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

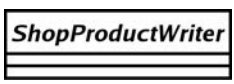
Łatwo się domyślić, że głównymi składnikami diagramów klas są same klasy. Klasa jest reprezentowana na diagramie prostokątem opatrzonym nazwą, jak na rysunku 6.1.



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 umieszczanego pomiędzy nawiasami klamrowymi — powinien on jedynie wyjaśniać warunki wymagane dla elementu.

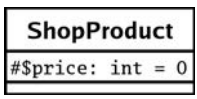
Interfejsy obrazuje się na diagramie klas tak samo jak klasy, tyle że należy je uzupełnić o stereotyp (element zapewniający rozszerzalność języka UML), jak to zrobiono na rysunku 6.4.

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*

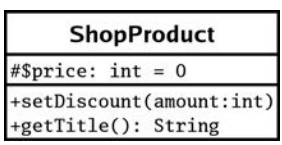
Symbol	Widoczność	Znaczenie
+	Publiczna	Atrybut dostępny ogólnie.
-	Prywatna	Atrybut dostępny wyłącznie w ramach bieżącej klasy.
#	Chroniona	Atrybut dostępny wyłącznie w ramach bieżącej klasy i jej pochodnych.

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*

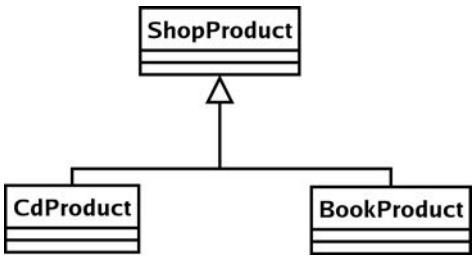
Jak widać, składnia operacji jest zbliżona do składni atrybutów. Nazwę metody poprzedza więc symbol jej widoczności. Operacje wyróżnia ujęta w nawiasy lista parametrów. Za listą parametrów, po znaku dwukropka, określany jest typ wartości zwracanej przez metodę (o ile jest zdefiniowany). Parametry na 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

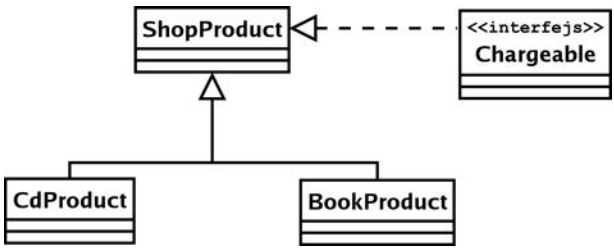
Język UML opisuje relację dziedziczenia jako relację uogólnienia, czyli „generalizacji" klas pochodnych w klasie bazowej[2]. Relacja ta jest reprezentowana na diagramie przez linię wiodącą od klasy pochodnej do klasy bazowej. Linia kończy się zarysem (niewypełnionym) strzałki.

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.

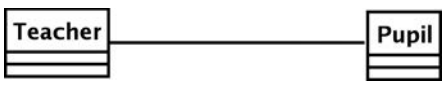


Rysunek 6.8. *Opis relacji implementacji*

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.

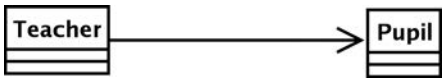


Rysunek 6.9. *Powiązanie*

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

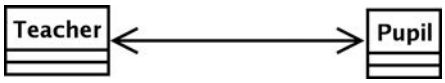
Taki opis nie informuje jednoznacznie o rodzaju powiązania. Wiadomo jedynie, że obiekt klasy `Teacher` będzie przechowywał referencję do jednego bądź wielu obiektów klasy `Pupil` albo odwrotnie — to obiekt klasy `Pupil` będzie przechowywał referencje obiektów klasy `Teacher`. Relacja powiązania może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



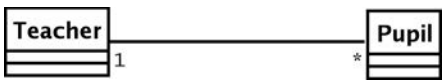
Rysunek 6.10. *Powiązanie jednokierunkowe*

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.



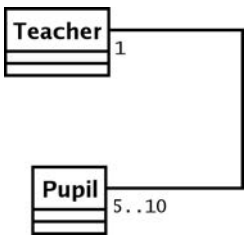
Rysunek 6.11. *Powiązanie dwukierunkowe*

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`.



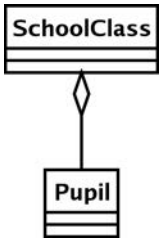
Rysunek 6.13. *Definiowanie krotności relacji powiązania*

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 wchodzą 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.

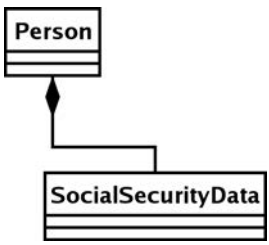
Rysunek 6.14 ilustruje dwie klasy: `SchoolClass` (grupa zajęciowa) i `Pupil` (uczeń). Klasa składa się tu z uczniów.



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.

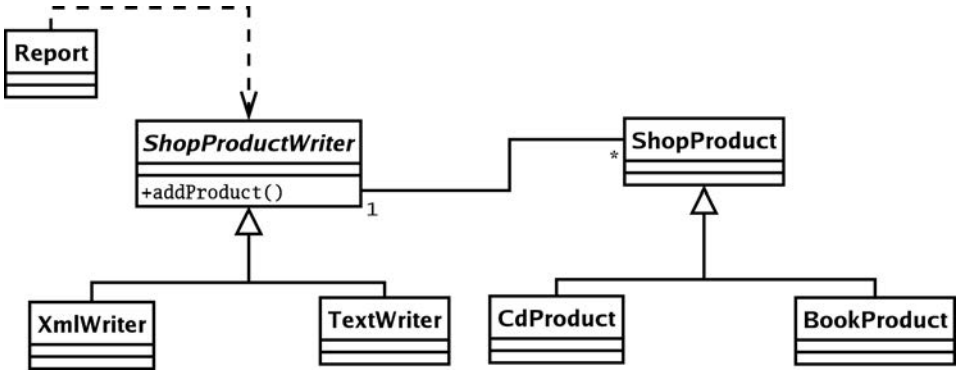
Obiekt klasy używanej może zostać przekazany do klasy używającej za pośrednictwem argumentu wywołania metody, może też zostać pozyskany jako wartość zwracana z wywołania metody.

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`.

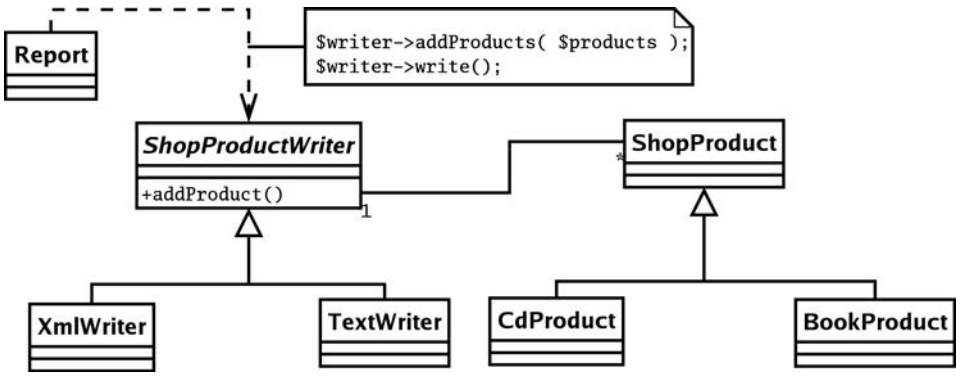
Notki

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

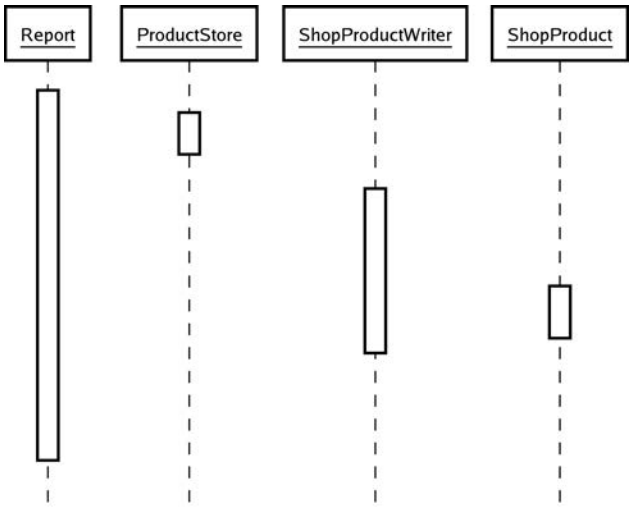
Diagram sekwencji operuje raczej obiektami niż klasami. Służy do modelowania poszczególnych etapów procesu przebiegającego w systemie.

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*

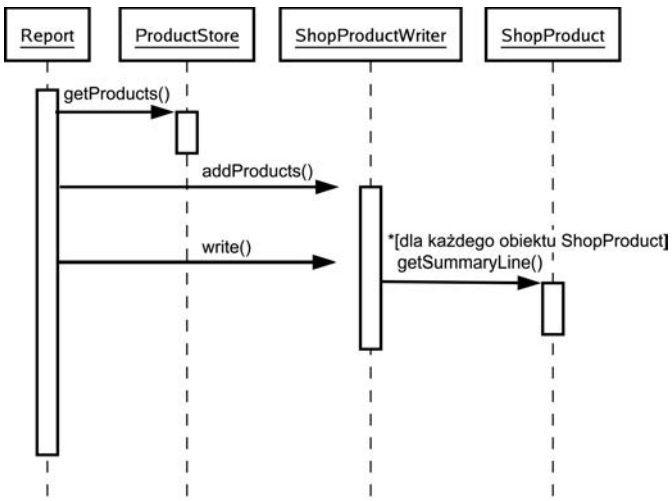
Obiekty oznaczyliśmy na diagramie nazwami ich klas. Gdyby w systemie działało niezależnie wiele egzemplarzy tej samej klasy, moglibyśmy umieścić na diagramie osobne bloki tych obiektów, stosując dla nich etykiety w formacie *`obiekt`*`::`*`klasa`* (np. `product1::ShopProduct`).

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



Rysunek 6.19. *„Linie życia" obiektów na diagramie sekwencyjnym*

Przerywane 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 dobierać 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

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ż

PHP. Obiekty, wzorce, narzędzia

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 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 obiektową rewolucję. Dostęp do zaawansowanych narzędzi, wzrost świadomości oraz zmiany w samym języku wystarczyły, by programiści coraz powszechniej zaczęli stosować techniki obiektowe 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, 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!

Helion

Nr katalogowy: 6208

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