IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG
TWÓJ KOSZYK
DODAJ DO KOSZYKA
CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK
CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

100 sposobów na PHP

Autor: Jack Herrington
Tłumaczenie: Radosław Meryk
ISBN: 83-246-0426-X
Tytuł oryginału: PHP Hacks
Format: B5, stron: 440

Zbiór rozwiązań dla twórców dynamicznych witryn WWW

- Korzystanie z danych pochodzących z innych witryn WWW
- Dynamiczne generowanie grafiki i animacji Flash
- Obsługa komunikatorów internetowych i protokołu IRC

Język PHP zdobył ogromną popularność jako narzędzie do tworzenia dynamicznych witryn WWW, a grono jego użytkowników stale się powiększa. Programiści i projektanci doceniają jego możliwości, szybkość i wygodę. Standardowe już zastosowania języka PHP — łączenie witryny WWW z bazą danych, przechowywanie treści artykułów w tabelach i obsługa formularzy nie wyczerpują możliwości tej platformy programistycznej. PHP oferuje znacznie więcej — pozwala między innymi na dynamiczne generowanie grafiki, korzystanie z usług sieciowych i protokołu SOAP oraz przetwarzanie plików XML.

Książka „100 sposobów na PHP" to coś więcej niż kolejny podręcznik tworzenie aplikacji WWW. Znajdziesz w niej mniej znane sposoby wykorzystywania PHP przy budowaniu witryn internetowych. Nauczysz się korzystać z biblioteki PEAR, tworzyć interfejsów użytkownika z wykorzystaniem języka DHTML oraz technologii SVG oraz generować pliki RTF, CSV i XLS. Dowiesz się, jak stosować wzorce projektowe i testować aplikacje wykorzystując testy jednostkowe. Poznasz zasady programowania obiektowego w PHP i tchniesz nowe życie w działające już aplikacje dodając do nich ciekawe „wodotryski", których przykłady znajdziesz w tej książce.

- Instalacja PHP oraz biblioteki PEAR
- Projektowanie interfejsów użytkownika
- Łączenie PHP z DHTML oraz JavaScript
- Generowanie grafiki bitmapowej i wektorowej
- Manipulowanie danymi w bazie za pomocą plików XML
- Łączenie aplikacji WWW z GoogleMaps oraz Wikipedią
- Wykorzystywanie wzorców projektowych
- Testowanie aplikacji
- Generowanie animacji Flash
- Wysyłanie SMS-ów oraz wiadomości na serwery IRC

Poznaj nietypowe zastosowania języka PHP

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

O autorach 9

Przedmowa 13

Rozdział 1. Instalacja i podstawy 21
1. Instalacja PHP 21
2. Instalacja modułów PEAR 32

Rozdział 2. Projektowanie aplikacji internetowych 35
3. Tworzenie interfejsu z wykorzystaniem „skórek” 35
4. Tworzenie nawigacji typu breadcrumb 39
5. Tworzenie ramek na stronach WWW 43
6. Zastosowanie zakładek w interfejsie aplikacji internetowych 47
7. Zapewnienie użytkownikom możliwości formatowania stron z wykorzystaniem techniki XSL 50
8. Tworzenie prostych wykresów HTML 53
9. Prawidłowe ustawianie rozmiaru znaczników graficznych 55
10. Wysyłanie wiadomości e-mail w formacie HTML 57

Rozdział 3. DHTML 61
11. Umieszczenie na stronie interaktywnego arkusza kalkulacyjnego 61
12. Tworzenie wyskakujących wskazówek 64
13. Tworzenie list w trybie przeciągnij i upuść 66
14. Tworzenie dynamicznych wykresów 69
15. Podział treści na rozwijane sekcje 74
16. Tworzenie rozwijanych „samoprzylepnych” karteczek 78
17. Tworzenie dynamicznych menu nawigacyjnych 80
18. Dynamiczne ukrywanie kodu JavaScript 83
19. Tworzenie zegara binarnego za pomocą kodu DHTML 85
20. Ułatwienie implementacji Ajax za pomocą modułu JSON 88
21. Utworzenie pokazu slajdów za pomocą kodu DHTML 91
22. Wykorzystanie grafiki wektorowej w PHP 93

23. Tworzenie narzędzia do wybierania kolorów ... 96
24. Tworzenie grafu łączy ... 98
25. Utworzenie interaktywnego kalendarza ... 101
26. Tworzenie efektu przewijania map Google ... 105

Rozdział 4. Grafika ... 113
27. Tworzenie miniaturek ... 113
28. Tworzenie atrakcyjnej grafiki za pomocą SVG ... 115
29. Uproszczenie obsługi grafiki dzięki wykorzystaniu obiektów ... 118
30. Podział obrazu na kilka mniejszych ... 126
31. Tworzenie wykresów w PHP ... 130
32. Nakładanie obrazów ... 132
33. Dostęp do zdjęć iPhoto z poziomu PHP ... 136

Rozdział 5. Bazy danych i XML ... 149
34. Projektowanie lepszego schematu SQL ... 149
35. Uniwersalny dostęp do bazy danych ... 154
36. Tworzenie dynamicznych obiektów dostępu do bazy danych ... 156
37. Generowanie instrukcji CRUD dla baz danych ... 160
38. Zastosowanie wyrażeń regularnych do łatwego czytania dokumentów XML ... 169
39. Eksportowanie schematu bazy danych w formacie XML ... 172
40. Prosty mechanizm obsługi zapytań do bazy danych w formacie XML ... 174
41. Generowanie kodu SQL ... 175
42. Generowanie kodu PHP dostępu do bazy danych ... 178
43. Konwersja CSV na PHP ... 184
44. Odczyt danych ze stron WWW ... 187
45. Odczytywanie danych z arkuszy Excela wgranych na serwer ... 192
46. Ładowanie danych z Excela do bazy danych ... 196
47. Przeszukiwanie dokumentów programu Microsoft Word ... 200
48. Dynamiczne tworzenie dokumentów RTF ... 203
49. Dynamiczne tworzenie arkuszy Excela ... 208
50. Tworzenie kolejki wiadomości ... 213

Rozdział 6. Projektowanie aplikacji ... 217
51. Tworzenie modularnych interfejsów ... 217
52. Obsługa tekstu Wiki ... 221
53. Przekształcanie dowolnych obiektów na tablice ... 224
54. Tworzenie prawidłowego kodu XML ... 227
55. Rozwiązanie problemu podwójnego przesyłania ... 230
56. Tworzenie spersonalizowanych raportów ... 234
57. Tworzenie systemu logowania ... 236

58. Zabezpieczenia z wykorzystaniem ról 241
59. Migracja do haseł MD5 248
60. Zastosowanie modułu mod_rewrite do tworzenia użytecznych adresów URL 252
61. Utworzenie mechanizmu przekierowania reklam 257
62. Wykorzystanie przycisku Buy Now serwisu PayPal 260
63. Odczytywanie informacji o lokalizacji użytkowników aplikacji 269
64. Import informacji z plików vCard 270
65. Tworzenie plików vCard na podstawie danych aplikacji 273
66. Tworzenie koszyka na zakupy 274

Rozdział 7. Wzorce projektowe 283
67. Obserwacja obiektów 284
68. Tworzenie obiektów z wykorzystaniem wzorca Fabryka Abstrakcyjna 287
69. Elastyczne tworzenie obiektów z wykorzystaniem wzorca Metoda Fabrykująca 290
70. Wyodrębnienie kodu konstrukcyjnego za pomocą wzorca Budowniczy 292
71. Oddzielenie części „co” od „jak” za pomocą wzorca Strategia 296
72. Łączenie dwóch modułów z wykorzystaniem wzorca Adapter 299
73. Pisanie przenośnego kodu z wykorzystaniem wzorca Most 302
74. Rozszerzalne przetwarzanie z wykorzystaniem wzorca Łańcuch odpowiedzialności 305
75. Podział rozbudowanych klas na mniejsze z wykorzystaniem wzorca Kompozyt 309
76. Uproszczenie interfejsu API z wykorzystaniem wzorca Fasada 311
77. Tworzenie stałych obiektów za pomocą wzorca Singleton 315
78. Ułatwienie wykonywania operacji z danymi dzięki zastosowaniu wzorca Wizytator 318

Rozdział 8. Testowanie 323
79. Testowanie kodu za pomocą testów jednostkowych 323
80. Generowanie własnych testów jednostkowych 325
81. Wyszukiwanie niesprawnych łączy 329
82. Testowanie aplikacji z wykorzystaniem symulowanych użytkowników 331
83. Testowanie aplikacji z wykorzystaniem robotów 335
84. Testowanie witryny za pomocą aplikacji typu „pająk” 339
85. Automatyczne generowanie dokumentacji 343

Rozdział 9. Alternatywne interfejsy użytkownika 347
86. Tworzenie map z wykorzystaniem systemu MapServer 347
87. Tworzenie interfejsów GUI z wykorzystaniem biblioteki GTk 357
88. Wysyłanie nagłówków RSS do komunikatorów za pomocą protokołu Jabber 360
89. Komunikacja z aplikacją internetową za pomocą IRC 367

90. Odczyt źródeł RSS na konsoli PSP 369
91. Wyszukiwanie w Google według słów kluczowych 372
92. Utworzenie nowego interfejsu witryny Amazon.com 378
93. Wysyłanie wiadomości SMS za pomocą komunikatorów 381
94. Generowanie animacji Flasha 385

Rozdział 10. Dla zabawy 395
95. Tworzenie własnych map Google 395
96. Tworzenie dynamicznych list odtwarzania 400
97. Utworzenie centrum wymiany plików multimedialnych 403
98. Sprawdzanie statusu gry sieciowej za pomocą skryptu PHP 408
99. Wikipedia na konsoli PSP 410
100. Gdzie jest lepsza pogoda? 417

Skorowidz 421

ROZDZIAŁ SIÓDMY

Wzorce projektowe

Sposoby 67. – 78.

W 1994 roku Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides opublikowali książkę pt. *Design Patterns*[1] (Addison-Wesley). Z powodu doskonałej treści publikacja bardzo szybko została zaliczona do klasyki literatury informatycznej. Jednym z jej osiągnięć było wypromowanie nowego metajęzyka w dziedzinie inżynierii i architektury systemów. Główna idea książki — opracowanie zbioru wzorców struktury obiektów — została zapożyczona z budownictwa i przystosowana do programowania.

Zbiór 40 wzorców projektowych zaprezentowanych w książce *Design Patterns* to efekt wielu lat doświadczeń. Każdy z nich opisano w formie neutralnej pod względem języka. Większość można zastosować w każdym środowisku projektowym.

Niektóre wzorce projektowe, na przykład *Iterator*, opracowano specjalnie po to, by uzupełnić braki języków programowania (w przypadku wzorca *Iterator* chodzi o język C++). W PHP jest wbudowana implementacja tego wzorca — konstrukcja **`foreach`**.

Wzorce projektowe nie były zbyt często wykorzystywane w PHP. Przed wydaniem PHP w wersji 5 język PHP i jego środowisko projektowe nie były traktowane w branży programistów poważnie. Obecnie, dzięki rozbudowanemu modelowi obiektowemu, dobrym środowiskom IDE i dużej popularności języka wśród programistów, branża zaczyna dostrzegać PHP. Zastosowanie wzorców projektowych w PHP opisano w niektórych najnowszych wydawnictwach poświęconych temu językowi. Uważam, że warto poświęcić im nieco uwagi także w tej książce.

W swej pracy postanowiłem odwoływać się do książki *Design Patterns*. Wybrałem z niej podzbiór wzorców do zaimplementowania. Wzorce te, wraz z ich implementacją, dają solidne podstawy architektury systemów. Mogą być także inspiracją do opracowywania własnego kodu.

[1] Polskie wydanie *Wzorce projektowe* — Wydawnictwa Naukowo-Techniczne 2005 — *przyp. tłum.*

SPOSÓB 67. Obserwacja obiektów

Zastosowanie wzorca Obserwator do luźnego wiązania obiektów.

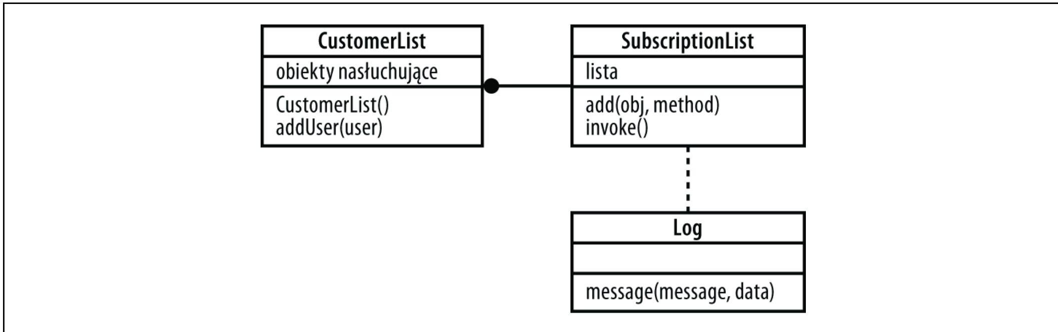
Luźne wiązanie obiektów (ang. *loose coupling*), pomimo że niewiele osób dobrze interpretuje ten termin, ma kluczowe znaczenie dla każdego projektu na dużą skalę. Czy zdarzyło się Wam wprowadzić niewielką modyfikację w projekcie, w której wyniku trzeba było zmienić w nim niemal wszystko? Taka sytuacja zdarza się nader często, a jej przyczyną jest ścisłe wiązanie pomiędzy modułami programu. Jeśli jeden z nich przestaje działać, podobnie dzieje się z resztą.

Wzorzec **Obserwator** rozluźnia powiązania pomiędzy obiektami dzięki zastosowaniu prostszego kontraktu. Obiekt może być obserwowany dzięki udostępnieniu mechanizmu rejestracji. W przypadku, gdy obserwowany obiekt się zmienia, informuje o tym obiekty obserwujące za pomocą obiektu powiadamiającego. Obserwowanego obiektu nie interesuje ani sposób, ani powód, dla którego jest obserwowany. Nie wie nawet tego, jakie typy obiektów go obserwują. Co więcej, obiektów obserwujących zazwyczaj nie interesuje sposób lub powód modyfikacji obiektu. Jedyne, co śledzą, to zmiany.

Klasycznym przykładem wzorca *Obserwator* jest kod obsługi okna dialogowego, który obserwuje stan pola wyboru. Dla pola wyboru nie ma znaczenia, czy obserwuje go jeden obiekt czy tysiąc. Jeśli zmieni się jego stan, po prostu wysyła informację. Z kolei dla okna dialogowego nie ma znaczenia sposób implementacji pola wyboru. Istotny jest tylko jego stan oraz uzyskanie powiadomienia w przypadku, gdy się zmieni.

W podrozdziale zademonstruję wzorzec *Obserwator* na przykładzie listy klientów, którą można obserwować. Obiekt reprezentuje tabelę klientów w bazie danych. W przypadku dodania nowych klientów obiekt `CustomerList` wysyła powiadomienie. Do zaimplementowania obserwacji obiekt `CustomerList` wykorzystuje klasę `SubscriptionList`. Obiekty nasłuchujące to egzemplarze klasy `SubscriptionList`, które inne obiekty wykorzystują do zarejestrowania się w obiekcie `CustomerList`. Obiekty te używają metody `add()` w celu dodania się do listy, natomiast obiekt `CustomerList` wykorzystuje metodę `invoke()` w celu wysłania komunikatu do obiektów nasłuchujących. Dla obiektu `CustomerObject` nie ma znaczenia, czy jest tysiąc obiektów nasłuchujących czy nie ma żadnego. Ciekawą własnością wzorca *Obserwator* jest fakt, iż obiekty nasłuchujące nie komunikują się w sposób bezpośredni ani nie zależą od obiektu `CustomerList`. Obiekty nasłuchujące są odizolowane od klientów za pomocą klasy `SubscriptionList`.

W pokazanym przykładzie zdefiniujemy jeden obiekt nasłuchujący: `Log`, którego działanie polega na wyświetlaniu na konsoli komunikatów przesyłanych przez obiekt `CustomerList`. Powiązania pomiędzy obiektami zaprezentowano na rysunku 7.1.



Rysunek 7.1. Obiekty CustomerList oraz związany z nim obiekt SubscriptionList wraz z obiektem Log

Kod

Kod pokazany na listingu 7.1 należy zapisać w pliku *observer.php*.

Listing 7.1. Przykład zastosowania wzorca projektowego Obserwator

```
<?php
class Log
{
  public function message( $sender, $messageType, $data )
  {
    print $messageType." - ".$data."\n";
  }
}

class SubscriptionList
{
  var $list = array();

  public function add( $obj, $method )
  {
    $this->list []= array( $obj, $method );
  }

  public function invoke()
  {
    $args = func_get_args();
    foreach( $this->list as $l ) { call_user_func_array( $l, $args ); }
  }
}

class CustomerList
{
  public $listeners;

  public function CustomerList()
  {
    $this->listeners = new SubscriptionList();
  }

  public function addUser( $user )
  {
    $this->listeners->invoke( $this, "add", "$user" );
  }
}
```

```
$l = new Log();
$cl = new CustomerList();
$cl->listeners->add( $l, 'message' );
$cl->addUser( "starbuck" );
?>
```

Wykorzystanie sposobu

Powyższy kod można uruchomić w wierszu polecenia w następujący sposób:

```
% php observer.php
add - starbuck
```

Kod najpierw tworzy listę klientów oraz obiekt `Log`. Następnie obiekt `Log` wykonuje subskrypcję listy klientów za pomocą metody `add()`. Ostatnia czynność to dodanie użytkownika do listy klientów. Powoduje to wysłanie wiadomości do obiektów nasłuchujących — w tym przypadku do obiektu `Log`, który wyświetla komunikat o dodaniu nowego klienta.

Bez trudu można rozszerzyć zaprezentowany kod i skonfigurować konto klienta po dodaniu lub, na przykład, wysłać e-mail do nowego użytkownika — obie te operacje można wykonać bez konieczności modyfikacji kodu obiektu `CustomerList`. Właśnie na tym polega luźne wiązanie obiektów i dlatego wzorzec projektowy *Obserwator* jest taki ważny.

Istnieje bardzo wiele zastosowań wzorca projektowego *Obserwator* w programowaniu. Wykorzystuje się go, między innymi, w systemach okienkowych do implementacji **mechanizmu zdarzeń** (ang. *events*). Niektóre firmy, na przykład Tibco, tworzą cały model działania swoich przedsiębiorstw w oparciu o wzorzec *Obserwator*. Wykorzystują go do łączenia dużych podsystemów funkcjonalnych, takich jak Kadry i Płace. W systemach baz danych wzorzec *Obserwator* można wykorzystać do wywoływania kodu związanego z wyzwalaczami, które uaktywniają się w przypadku, gdy w bazie danych zostaną zmodyfikowane pewne typy rekordów. Mechanizm wykorzystujący wzorzec *Obserwator* przydaje się również w sytuacjach, gdy mamy świadomość, że zmiana stanu będzie istotna, ale jeszcze nie wiemy, gdzie te informacje wykorzystamy. Obiekty nasłuchujące można zaimplementować później i nie trzeba ich wiązać z obserwowanym obiektem.

Potencjalnym problemem wzorca projektowego *Obserwator* są pętle nieskończone. Mogą się zdarzyć, gdy obiekty obserwujące system jednocześnie go modyfikują. Na przykład rozwijane pole kombi modyfikuje wartość i informuje o tym strukturę danych. Struktura danych powiadamia rozwijane pole kombi, że wartość się zmieniła. Wtedy rozwijane pole kombi modyfikuje swoją wartość i wysyła kolejne powiadomienie do struktury danych itd. Najprostszym sposobem rozwiązania tego problemu jest wykluczenie wystąpienia rekurencji w kodzie obsługi pola kombi. Obiekt powinien zignorować komunikat od struktury danych, jeśli właśnie powiadamia strukturę danych o swojej nowej wartości.

Zobacz też

- „Przekształcanie dowolnych obiektów na tablice” [Sposób 53.].
- „Tworzenie kolejki wiadomości” [Sposób 50.].

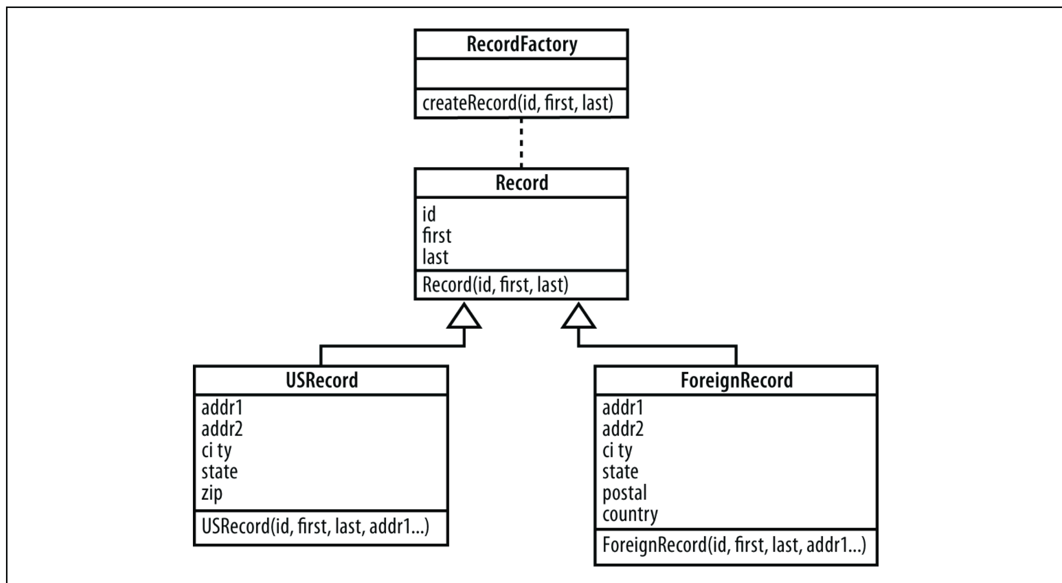
SPOSÓB 68.

Tworzenie obiektów z wykorzystaniem wzorca Fabryka Abstrakcyjna

Wykorzystanie wzorca projektowego *Fabryka Abstrakcyjna* do śledzenia typu tworzonych obiektów.

Wzorzec projektowy *Fabryka Abstrakcyjna* (ang. *Abstract Factory*) to maszyna do produkcji wzorców projektowych. Wystarczy zdefiniować to, czego chcemy, a wzorzec zadba o utworzenie obiektów na podstawie wprowadzonych kryteriów. Zaletą wzorca jest możliwość modyfikacji typu tworzonych obiektów poprzez modyfikację „fabryki”.

W prostym przykładzie zaprezentowanym w tym podrozdziale utworzymy obiekty `Record`, z których każdy będzie miał swój identyfikator, imię i nazwisko. Związki pomiędzy poszczególnymi klasami pokazano na rysunku 7.2.



Rysunek 7.2. Klasy Record i RecordFactory

Obiekty-fabryki często tworzą więcej niż jeden typ obiektów. Dla uproszczenia przykładu ograniczyłem obiekt-fabrykę do tworzenia tylko jednego typu obiektów.

W PHP nie można rygorystycznie wymusić tworzenia obiektów określonego typu wyłącznie przez obiekt-fabrykę. Jeśli jednak będziemy stosowali obiekt-fabrykę stosunkowo często, inżynierowie kopiując i wklejając nasz kod, będą w efekcie stosowali obiekt-fabrykę. Szybko stanie się on standardem de facto tworzenia różnych typów obiektów.

Kod

Kod pokazany na listingu 7.2 zapiszemy w pliku *abs_factory.php*.

Listing 7.2. Zastosowanie wzorca projektowego Fabryka Abstrakcyjna

```
<?php
class Record
{
  public $id = null;
  public $first = null;
  public $last = null;

  public function __construct( $id, $first, $last )
  {
    $this->id = $id;
    $this->first = $first;
    $this->last = $last;
  }
}

class USRecord extends Record
{
  public $addr1 = null;
  public $addr2 = null;
  public $city = null;
  public $state = null;
  public $zip = null;

  public function __construct( $id, $first, $last,
    $addr1, $addr2, $city, $state, $zip )
  {
     parent::__construct( $id, $first, $last );
   $this->addr1 = $addr1;
   $this->addr2 = $addr2;
   $this->city = $city;
   $this->state = $state;
   $this->zip = $zip;
  }
}

class ForeignRecord extends Record
{
  public $addr1 = null;
  public $addr2 = null;
  public $city = null;
  public $state = null;
  public $postal = null;
  public $country = null;

  public function __construct( $id, $first, $last,
    $addr1, $addr2, $city, $state, $postal, $country )
  {
     parent::__construct( $id, $first, $last );
   $this->addr1 = $addr1;
   $this->addr2 = $addr2;
   $this->city = $city;
   $this->state = $state;
   $this->postal = $postal;
   $this->country = $country;
  }
}
```

```
class RecordFactory
{
  public static function createRecord( $id, $first, $last,
    $addr1, $addr2, $city, $state, $postal, $country )
  {
  if ( strlen( $country ) > 0 && $country != "USA" )
      return new ForeignRecord( $id, $first, $last,
        $addr1, $addr2, $city, $state, $postal, $country );
  else
      return new USRecord( $id, $first, $last,
        $addr1, $addr2, $city, $state, $postal );
  }
}

function readRecords()
{
  $records = array();

  $records []= RecordFactory::createRecord(
   1, "Jack", "Herrington", "4250 San Jaquin Dr.", "",
   "Los Angeles", "CA", "90210", ""
  );
  $records []= RecordFactory::createRecord(
   1, "Maria", "Kowalska", "Pstrowskiego 4", "",
   "Malbork", "pomorskie", "82-200", "Polska"
  );

  return $records;
}

$records = readRecords();
foreach( $records as $r )
{
  $class = new ReflectionClass( $r );
  print $class->getName()." - ".$r->id." - ".$r->first." - ".$r->last."\n";
}
?>
```

W pierwszej części kodu zaimplementowano klasę bazową `Record` oraz klasy pochodne `USRecord` i `ForeignRecord`. Są to stosunkowo proste klasy opakowujące dla struktur danych. Klasa-fabryka może tworzyć zarówno obiekty `USRecord`, jak `ForeignRecord` w zależności od danych, które zostaną do niej przekazane. Kod testujący na końcu skryptu dodaje kilka rekordów, po czym wyświetla ich typ oraz niektóre dane.

Wykorzystanie sposobu

Do uruchomienia przykładu zastosujemy interpreter PHP działający w wierszu polecenia w następujący sposób:

```
% php abs_factory.php
USRecord - 1 - Jack - Herrington
ForeignRecord - 1 - Maria - Kowalska
```

W aplikacji bazodanowej w PHP można zastosować wzorzec projektowy *Fabryka Abstrakcyjna* na kilka sposobów:

Tworzenie obiektu bazy danych
: Obiekt-fabryka tworzy wszystkie typy obiektowe powiązane z poszczególnymi tabelami w bazie danych.

Tworzenie przenośnych obiektów
: Obiekt-fabryka tworzy różne obiekty w zależności od typu systemu operacyjnego, w którym działa kod, bądź od typów baz danych, z którymi aplikacja się łączy.

Tworzenie według standardu
: Aplikacja obsługuje różnorodne standardy formatów plików i wykorzystuje obiekt-fabrykę do tworzenia obiektów odpowiednich dla poszczególnych typów plików. Obiekty czytające pliki mogą się zarejestrować w obiekcie-fabryce w celu dodania obsługi plików bez konieczności modyfikacji klientów.

Wykorzystywanie wzorców projektowych przez pewien czas pozwala programiście na uzyskanie wyczucia co do tego, kiedy warto zastosować określony wzorzec. Wzorzec *Fabryka Abstrakcyjna* stosuje się w przypadku tworzenia dużej liczby obiektów różnych typów. Jak można się przekonać, zmiany typów tworzonych obiektów lub sposobu ich tworzenia często powodują konieczność wielu modyfikacji w kodzie. W przypadku zastosowania klasy-fabryki zmianę trzeba wprowadzić tylko w jednym miejscu.

Zobacz też

- „Elastyczne tworzenie obiektów z wykorzystaniem wzorca Metoda Fabrykująca" [Sposób 69.].

SPOSÓB 69.

Elastyczne tworzenie obiektów z wykorzystaniem wzorca Metoda Fabrykująca

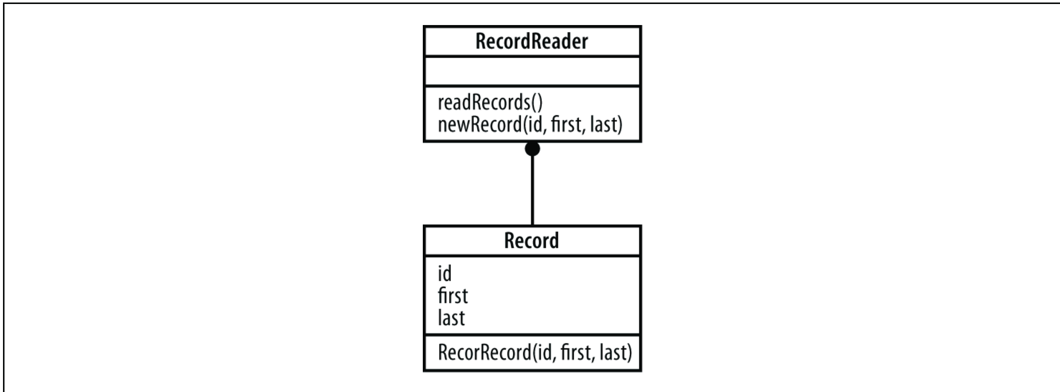
Wykorzystanie wzorca *Metoda Fabrykująca* podczas tworzenia obiektów w celu umożliwienia klasom pochodnym modyfikacji typów tworzonych obiektów.

Z wzorcem *Fabryka Abstrakcyjna* jest blisko związany wzorzec *Metoda Fabrykująca*. Jego działanie jest dość oczywiste. Jeśli mamy klasę, która tworzy dużą liczbę obiektów, możemy wykorzystać metody chronione hermetyzujące operacje tworzenia. W ten sposób klasy pochodne, w celu utworzenia różnych typów obiektów, mogą przesłonić metody chronione klasy-fabryki.

W pokazanym przykładzie klasa `RecordReader` zamiast skorzystania z klasy-fabryki wykorzystuje metodę `NewRecord()`. W ten sposób klasy pochodne klasy `RecordReader` mogą modyfikować typ tworzonych obiektów `Record` poprzez przesłonięcie metody `newRecord()`. Sytuację tę graficznie przedstawiono na rysunku 7.3.

Kod

Kod pokazany na listingu 7.3 zapiszemy w pliku *factory_method.php*.



Rysunek 7.3. Związki pomiędzy klasami RecordReader i Record

Listing 7.3. Przykład metod fabrycznych klasy

```
<?php
class Record
{
  public $id = null;
  public $first = null;
  public $last = null;

  public function Record( $id, $first, $last )
  {
    $this->id = $id;
    $this->first = $first;
    $this->last = $last;
  }
}

class RecordReader
{
  function readRecords()
  {
    $records = array();

    $records []= $this->newRecord( 1, "Jack", "Herrington" );
    $records []= $this->newRecord( 2, "Lori", "Herrington" );
    $records []= $this->newRecord( 3, "Megan", "Herrington" );

    return $records;
  }
  protected function newRecord( $id, $first, $last )
  {
    return new Record( $id, $first, $last );
  }
}

$rr = new RecordReader();
$records = $rr->readRecords();
foreach( $records as $r )
{
  print $r->id." - ".$r->first." - ".$r->last."\n";
}
?>
```

Wykorzystanie sposobu

Zaprezentowany kod uruchamia się w wierszu polecenia w następujący sposób:

```
%php factory_method.php
1 - Jack - Herrington
2 - Lori - Herrington
3 - Megan - Herrington
```

Po utworzeniu egzemplarza obiektu `RecordReader` następuje wywołanie jego metody `readRecords()`, która z kolei wywołuje metodę `newRecord` w celu utworzenia wszystkich obiektów `Record`. Utworzone obiekty są następnie wyświetlane na konsoli za pomocą pętli **`foreach`**.

W najbardziej widoczny sposób wzorzec *Metoda Fabrykująca* zastosowano w interfejsie API XML DOM organizacji W3C instalowanym w ramach bazowej instalacji PHP 5. Obiekt `DOMDocument`, który spełnia rolę korzenia każdego drzewa DOM, zawiera zbiór metod-fabryk: `createElement()`, `createAttrribute()`, `createTextNode()` itd. Implementacje pochodne od obiektu `DOMDocument` mogą przesłaniać te metody w celu zmiany obiektów tworzonych podczas ładowania drzew XML z dysku, zmiennych tekstowych lub tworzonych „w locie".

Podobnie jak w przypadku wzorca *Fabryka Abstrakcyjna* najważniejszą przesłanką do wykorzystania wzorca *Metoda Fabrykująca* jest sytuacja, gdy piszemy dużo kodu tworzącego obiekty. Dzięki zastosowaniu wzorca *Fabryka Abstrakcyjna* bądź *Method Factory* zyskujemy pewność, że jeśli zmieni się typ obiektów, który chcemy tworzyć, lub sposób ich tworzenia, zmiany w kodzie będą minimalne.

Zobacz też

- „Tworzenie obiektów z wykorzystaniem wzorca Fabryka Abstrakcyjna" [Sposób 68.].

SPOSÓB 70. Wyodrębnienie kodu konstrukcyjnego za pomocą wzorca Budowniczy

Wykorzystanie wzorca *Budowniczy* do wyodrębnienia kodu, który wykonuje rutynowe operacje konstrukcyjne, takie jak tworzenie dokumentów HTML lub tekstu wiadomości e-mail.

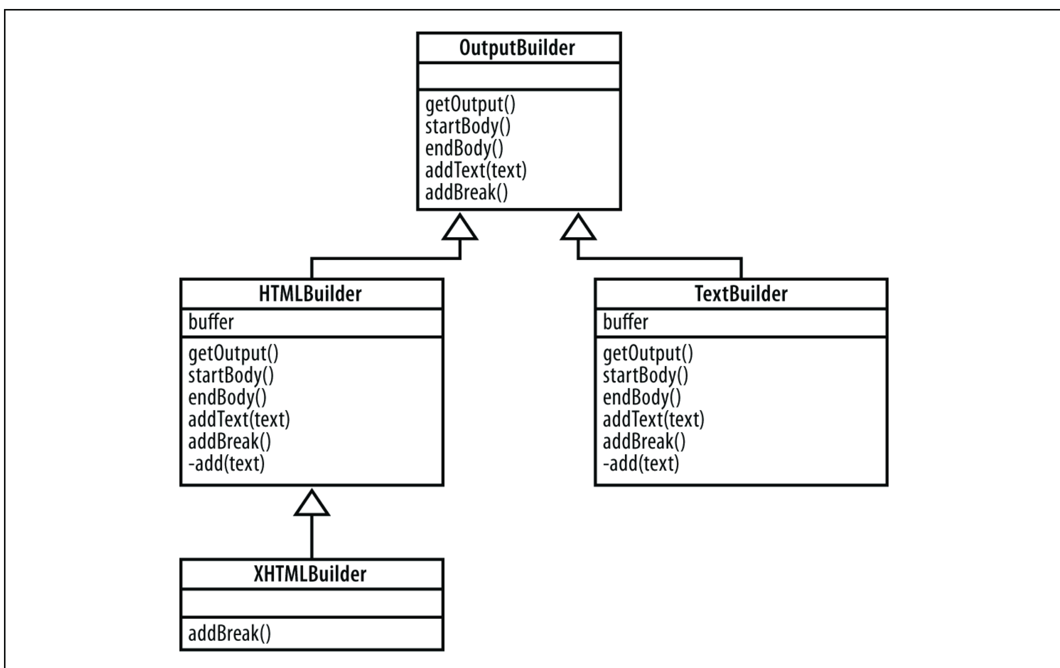
Wielokrotnie odnoszę wrażenie, że kod, który coś tworzy, jest najbardziej elegancki w całym systemie. Myślę, że jest tak dlatego, że poświęciłem rok na pisanie książki o generowaniu kodu, która w całości jest poświęcona kodowi konstrukcyjnemu.

Chciałbym dodać, że książka *Code Generation in Action* jest ciągle dostępna i może być doskonałym prezentem świątecznym dla przyjaciół lub członków rodziny.

Przykładem kodu konstrukcyjnego może być kod odczytujący dokument XML z dysku i tworzący jego reprezentację w pamięci. Innym może być moduł tworzący wiadomości e-mail przypominające klientom o tym, że upłynął termin płatności.

W niniejszym podrozdziale pokażę przykład tworzenia wiadomości o spóźnionych płatnościach. Zrobię to jednak sposobem: wykorzystam wzorzec *Budowniczy*, dzięki czemu kod tworzący wiadomość w formacie HTML będzie można wykorzystać do tworzenia wiadomości w formacie XHTML lub tekstowym.

W kodzie, który pisze wiadomość o spóźnionej płatności, zamierzam wykorzystać obiekt-konstruktora zamiast bezpośredniego tworzenia ciągu znaków. Obiekt ten będzie zawierał szereg metod, tak jak pokazano na rysunku 7.4. Kod tworzący wiadomość jest umieszczony pomiędzy wywołaniami metod `startBody()` oraz `endBody()`. Metoda `addText()` dodaje tekst wiadomości, natomiast `addBreak()` — znak zakończenia wiersza.



Rysunek 7.4. Hierarchia obiektów tworzących wiadomości

Klasa abstrakcyjna `OutputBuilder` ma kilka zmaterializowanych egzemplarzy. Jednym z nich jest `HTMLBuilder` tworzący kod HTML. Klasą jej pochodną jest `XHTMLBuilder` — klasa modyfikująca działanie klasy nadrzędnej w sposób wystarczający do utworzenia wyniku zgodnego z XHTML-em. Ostatnią klasą jest `TextBuilder`, która tworzy reprezentację wiadomości w formacie zwykłego tekstu.

Kod

Kod pokazany na listingu 7.4 zapiszemy w pliku *builder.php*.

Listing 7.4. Zbiór przykładowych klas konstrukcyjnych i kod testowy

```
<?php
abstract class OutputBuilder
{
  abstract function getOutput();
  abstract function startBody();
  abstract function endBody();
  abstract function addText( $text );
  abstract function addBreak();
}

class HTMLBuilder extends OutputBuilder
{
  private $buffer = "";

  public function getOutput()
  {
    return "<html>\n".$this->buffer."\n</html>\n";
  }
  public function startBody() { $this->add( "<body>" ); }
  public function endBody() { $this->add( "</body>" ); }
  public function addText( $text ) { $this->add( $text ); }
  public function addBreak() { $this->add( "<br>\n" ); }

  protected function add( $text ) { $this->buffer .= $text; }
}

class XHTMLBuilder extends HTMLBuilder
{
  public function addBreak() { $this->add( "<br />\n" ); }
}

class TextBuilder extends OutputBuilder
{
  private $buffer = "";

  public function getOutput()
  {
    return $this->buffer."\n";
  }
  public function startBody() { }
  public function endBody() { }
  public function addText( $text ) { $this->add( $text ); }
  public function addBreak() { $this->add( "\n" ); }

  protected function add( $text ) { $this->buffer .= $text; }
}

function buildDocument( $builder )
{
  $builder->startBody();
  $builder->addText( 'Jack,' );
  $builder->addBreak();
  $builder->addText( 'Jesteś nam winien 10 000 zł. Życzymy MIŁEGO dnia.' );
  $builder->endBody();
}
```

```
print "HTML:\n\n";

$html = new HTMLBuilder();
buildDocument( $html );
echo( $html->getOutput() );

print "\nXHTML:\n\n";

$xhtml = new XHTMLBuilder();
buildDocument( $xhtml );
echo( $xhtml->getOutput() );

print "\nTekst:\n\n";

$text = new TextBuilder();
buildDocument( $text );
echo( $text->getOutput() );
?>
```

Wykorzystanie sposobu

Do uruchomienia kodu wykorzystamy interpreter PHP działający w wierszu polecenia:

```
% php builder.php
HTML:

<html>
<body>Jack,<br>
Jesteś nam winien 10 000 zł. Życzymy MIŁEGO dnia.</body>
</html>

XHTML:

<html>
<body>Jack,<br />
Jesteś nam winien 10 000 zł. Życzymy MIŁEGO dnia.</body>
</html>

Tekst:

Jack,
Jesteś nam winien 10 000 zł. Życzymy MIŁEGO dnia.
```

Wyświetlił się wynik działania trzech obiektów konstrukcyjnych. Pierwszy to wersja wiadomości w formacie HTML z prawidłowymi znacznikami HTML oraz znacznikiem `<br>`. Kod konstrukcyjny dla XHTML-a nieco zmodyfikował wiadomość — przekształcił znacznik `<br>` na `<br />`. Wersja tekstowa to po prostu zwykły tekst. Znak końca wiersza zastąpiono znakiem powrotu karetki.

Na początku kodu znajduje się definicja klasy abstrakcyjnej `OutputBuilder`, za którą występują poszczególne egzemplarze klas dla różnych formatów wyniku. Obiekt konstruktora wykorzystano w funkcji `buildDocument()`, która tworzy wiadomość. Kod na końcu skryptu to testy funkcji `buildDocument()` dla każdego z typów obiektów konstrukcyjnych.

Wzorzec *Budowniczy* w aplikacji internetowej w PHP można wykorzystać w kilku miejscach:

Odczyt plików
: W operacjach przetwarzania plików można wykorzystać wzorzec *Budowniczy* do oddzielenia operacji analizy treści pliku od tworzenia struktur danych w pamięci z danymi z pliku.

Zapis plików
: Zgodnie z tym, co pokazałem w tym podrozdziale, wzorzec *Budowniczy* można wykorzystać do tworzenia wielu formatów wynikowych za pomocą jednego systemu tworzenia dokumentów.

Generowanie kodu
: Wzorzec *Budowniczy* można zastosować do generowania kodu w wielu językach za pomocą jednego systemu generującego.

W środowisku .NET wykorzystuje się wzorzec *Budowniczy* do tworzenia kodu HTML strony wynikowej, tak aby za pomocą tej samej konstrukcji sterującej generować różne odmiany kodu HTML w zależności od typu przeglądarki żądającej strony.

SPOSÓB 71.

Oddzielenie części „co” od „jak” za pomocą wzorca Strategia

Wykorzystanie wzorca *Strategia* w celu oddzielenia kodu przeglądającego struktury danych od kodu, który je przetwarza.

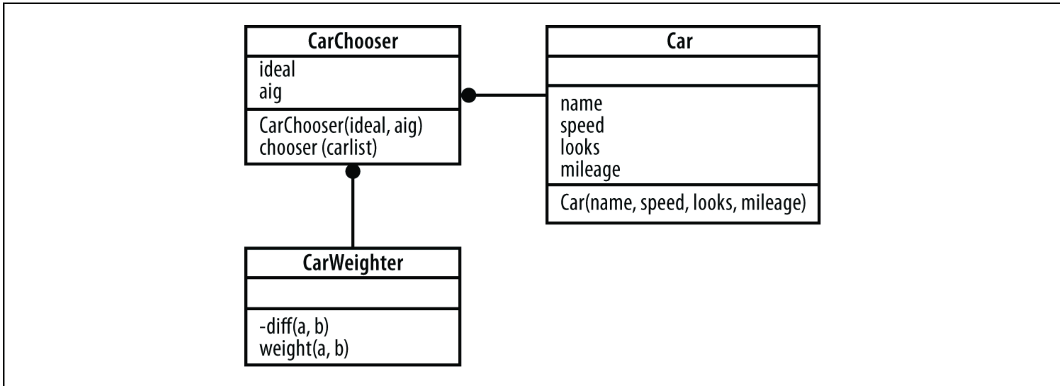
Wzorzec projektowy *Strategia* można wykorzystać do wyodrębnienia kodu przetwarzającego obiekty. Pozwala to na uniezależnienie sposobu przetwarzania kodu od jego lokalizacji.

W podrozdziale posłużę się aplikacją do wyboru samochodu. Skrypt będzie polecał samochód na podstawie wprowadzonych kryteriów wyszukiwania. W przykładzie wprowadzę specyfikację samochodu idealnego, a kod wybierze egzemplarz, który najbardziej pasuje do moich marzeń. Wielką zaletą wzorca *Strategia* jest możliwość modyfikacji kodu porównującego samochody w sposób niezależny od kodu wybierającego samochód.

Diagram UML dla sposobu pokazanego w tym podrozdziale pokazano na rysunku 7.5. Obiekt `CarChooser` wykorzystuje obiekt `CarWeighter` w celu porównania każdego z samochodów z idealnym modelem. Następnie skrypt zwraca do klienta najlepszy samochód.

Kod

Kod pokazany na listingu 7.5 zapiszemy w pliku *strategy.php*.



Rysunek 7.5. Relacje pomiędzy obiektami CarChooser, CarWeighter i Car

Listing 7.5. Zastosowanie wzorca Strategia

```
<?php
class Car
{
  public $name;
  public $speed;
  public $looks;
  public $mileage;
  public function Car( $name, $speed, $looks, $mileage )
  {
    $this->name = $name;
    $this->speed = $speed;
    $this->looks = $looks;
    $this->mileage = $mileage;
  }
}

class CarWeighter
{
  private function diff( $a, $b )
  {
    return abs( $a - $b );
  }

  public function weight( $a, $b )
  {
    $d = 0;
    $d += $this->diff( $a->speed, $b->speed );
    $d += $this->diff( $a->looks, $b->looks );
    $d += $this->diff( $a->mileage, $b->mileage );
    return ( 0 - $d );
  }
}

class CarChooser
{
  private $ideal;
  private $alg;

  function CarChooser( $ideal, $alg )
  {
```

```
    $this->ideal = $ideal;
    $this->alg = $alg;
  }

  public function choose( $carlist )
  {
    $minrank = null;
    $found = null;
    $alg = $this->alg;

    foreach( $carlist as $car )
    {
      $rank = $alg->weight( $this->ideal, $car );
      if ( !isset( $minrank ) ) $minrank = $rank;
      if ( $rank >= $minrank )
      {
        $minrank = $rank;
        $found = $car;
      }
    }

    return $found;
  }
}

function pickCar( $car )
{
  $carlist = array();
  $carlist []= new Car( "rakieta", 90, 30, 10 );
  $carlist []= new Car( "rodzinny", 45, 30, 55 );
  $carlist []= new Car( "ładny", 40, 90, 10 );
  $carlist []= new Car( "ekonomiczny", 40, 40, 90 );

  $cw = new CarWeighter();
  $cc = new CarChooser( $car, $cw );
  $found = $cc->choose( $carlist );
  echo( $found->name."\n" );
}

pickCar( new Car( "idealny", 80, 40, 10 ) );
pickCar( new Car( "idealny", 40, 90, 10 ) );
?>
```

Na początku skryptu zdefiniowałem klasę `Car` zawierającą nazwę samochodu oraz oceny dla szybkości, wyglądu i przebiegu. Każda z ocen mieści się w zakresie od 0 do 100 (głównie dlatego, aby obliczenia były proste). Następnie umieściłem definicję klasy `CarWeighter`, która porównuje dwa samochody i zwraca ocenę porównania. Na końcu zdefiniowałem klasę `CarChooser` wykorzystującą klasę `CarWeighter` do wyboru najlepszego samochodu na podstawie pewnych kryteriów wejściowych. Funkcja `pickCar()` tworzy zbiór samochodów, a następnie wykorzystuje obiekt `CarChooser` do wyboru z listy samochodu, który najlepiej spełnia kryteria (przekazane za pomocą obiektu `Car`).

Kod testowy umieszczony na końcu skryptu to żądanie wyboru dwóch samochodów — jednego, który ma wysoką ocenę szybkości i drugiego, który ładnie wygląda.

Wykorzystanie sposobu

Do uruchomienia kodu wykorzystamy interpreter PHP działający w wierszu polecenia:

```
% php strategy.php
rakieta
ładny
```

Z uzyskanego wyniku widać, że samochód, jaki aplikacja poleca mi w przypadku, gdy chodzi mi o szybkość, nazywa się *rakieta* (doskonałe określenie). W przypadku, gdy interesuje mnie coś bardziej seksownego, aplikacja proponuje samochód *ładny* — świetnie!

Kod, który wyciąga wniosek dotyczący tego, czy samochód spełnia kryteria, jest całkowicie oddzielony od kodu, który przeszukuje listę samochodów i wybiera z niej jeden pojazd. Algorytm porównujący samochód z wprowadzonymi kryteriami można zmodyfikować niezależnie od kodu wybierającego samochód z posortowanej listy. Na przykład w algorytmie porównującym samochody można uwzględnić marki, którymi ostatnio interesowaliśmy się, lub te, których w ostatnim czasie byliśmy posiadaczami. Można również zmodyfikować kod wybierający samochody tak, by proponował trzy z początku listy. W ten sposób użytkownik miałby dodatkowe możliwości wyboru.

SPOSÓB 72.

Łączenie dwóch modułów z wykorzystaniem wzorca Adapter

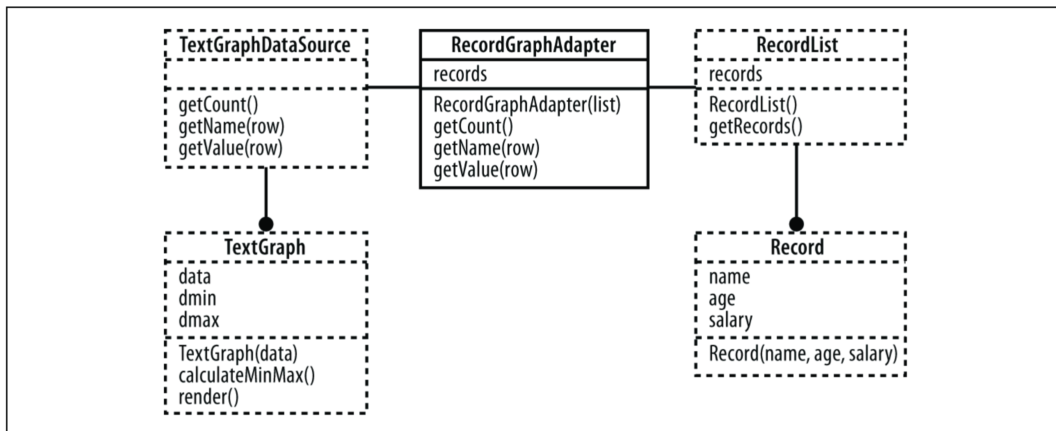
Wykorzystanie klasy-adaptera do przenoszenia danych pomiędzy dwoma modułami w sytuacji, gdy nie chcemy modyfikować interfejsu API żadnego z modułów.

Czasami trzeba pobrać dane z dwóch obiektów, z których każdy wykorzystuje inny format. Modyfikacja jednego lub drugiego formatu nie wchodzi w rachubę, ponieważ powodowałaby konieczność wprowadzania wielu dodatkowych zmian w pozostałej części kodu. Jednym z rozwiązań tego problemu jest wykorzystanie klasy-adaptera. Jest to klasa, która potrafi interpretować obie strony transmisji danych i przystosowuje jeden obiekt do komunikacji z drugim.

Klasa adapter zademonstrowana w tym podrozdziale przystosowuje dane pochodzące z fikcyjnej bazy danych do wykorzystania przez mechanizm tworzenia wykresów tekstowych.

Na rysunku 7.6 pokazano obiekt `RecordGraphAdapter` umieszczony pomiędzy obiektem `TextGraph` po lewej stronie a obiektem `RecordList` po prawej. Obiekt `TextGraph` w czytelny sposób specyfikuje format danych za pomocą klasy abstrakcyjnej `TextDataSource`. `RecordList` to klasa-kontener zawierająca listę obiektów `Record`. W każdym z nich są zapisane dane dotyczące nazwiska (`name`), wieku (`age`) i pensji (`salary`).

W pokazanym przykładzie utworzymy wykres pensji. Zadaniem klasy-adaptera jest pobranie danych z obiektu `RecordList` i przekształcenie ich na postać możliwą do przetworzenia przez obiekt `TextGraph`. W tym celu dane zostaną zapisane jako obiekty typu `TextGraphDataSource`.



Rysunek 7.6. Adapter umieszczony pomiędzy kodem tworzącym wykresy a danymi

Kod

Kod pokazany na listingu 7.6 zapiszemy w pliku *adapter.php*.

Listing 7.6. Przykład wykorzystania wzorca Adapter do tworzenia tekstowego wykresu

```
<?php
abstract class TextGraphDataSource
{
  abstract function getCount();
  abstract function getName( $row );
  abstract function getValue( $row );
}

class TextGraph
{
  private $data;
  private $dmin;
  private $dmax;

  public function TextGraph( $data )
  {
    $this->data = $data;
  }

  protected function calculateMinMax()
  {
    $this->dmin = 100000;
    $this->dmax = -100000;
    for( $r = 0; $r < $this->data->getCount(); $r++ )
    {
      $v = $this->data->getValue( $r );
      if ( $v < $this->dmin ) { $this->dmin = $v; }
      if ( $v > $this->dmax ) { $this->dmax = $v; }
    }
  }

  public function render()
  {
    $this->calculateMinMax();
    $ratio = 40 / ( $this->dmax - $this->dmin );
```

```
    for( $r = 0; $r < $this->data->getCount(); $r++ )
    {
      $n = $this->data->getName( $r );
      $v = $this->data->getValue( $r );
      $s = ( $v - $this->dmin ) * $ratio;
      echo( sprintf( "%10s : ", $n ) );
      for( $st = 0; $st < $s; $st++ ) { echo("*"); }
      echo( "\n" );
    }
  }
}

class Record
{
  public $name;
  public $age;
  public $salary;
  public function Record( $name, $age, $salary )
  {
    $this->name = $name;
    $this->age = $age;
    $this->salary = $salary;
  }
}

class RecordList
{
  private $records = array();

  public function RecordList()
  {
    $this->records []= new Record( "Janusz", 23, 26000 );
    $this->records []= new Record( "Beata", 24, 29000 );
    $this->records []= new Record( "Stefania", 28, 42000 );
    $this->records []= new Record( "Jerzy", 28, 120000 );
    $this->records []= new Record( "Grzegorz", 43, 204000 );
  }

  public function getRecords()
  {
    return $this->records;
  }
}

class RecordGraphAdapter extends TextGraphDataSource
{
  private $records;

  public function RecordGraphAdapter( $rl )
  {
    $this->records = $rl->getRecords();
  }
  public function getCount( )
  {
    return count( $this->records );
  }
  public function getName( $row )
  {
    return $this->records[ $row ]->name;
  }
  public function getValue( $row )
  {
    return $this->records[ $row ]->salary;
  }
}
```

```
$rl = new RecordList();

$ga = new RecordGraphAdapter( $rl );

$tg = new TextGraph( $ga );
$tg->render();
?>
```

Początek skryptu to kod odpowiedzialny za tworzenie wykresu. Zdefiniowano w nim klasę abstrakcyjną `TextGraphDataSource` oraz klasę `TextGraph` wykorzystującą klasę `TextGraphDataSource` jako format danych. W środkowej części skryptu zdefiniowano klasy `Record` i `RecordList` (zawierające dane do utworzenia wykresu). W trzeciej części zdefiniowano klasę `RecordGraphAdapter`, która przystosowuje klasę `RecordList` do wykorzystania jako źródło danych wykresu.

Kod testowy na początku skryptu najpierw tworzy obiekt `RecordList`, a następnie obiekt-adapter oraz obiekt `TextGraph`, który odwołuje się do adaptera. Wykres tworzy się poprzez odczyt danych z adaptera.

Wykorzystanie sposobu

Do uruchomienia kodu wykorzystamy interpreter PHP działający w wierszu polecenia:

```
% php adapter.php
    Janusz :
     Beata : *
  Stefania : ****
     Jerzy : **********************
  Grzegorz : ****************************************
```

Najmniej zarabia Janusz, a najwięcej Grzegorz. Na wykresie zastosowano automatyczne skalowanie, dlatego obok Janusza nie ma gwiazdek (minimum), natomiast obok Grzegorza wyświetla się 40 gwiazdek (maksimum). Świetnie ci idzie, Grzegorz! Ważniejsze w tym kodzie jest jednak to, że konwersja danych przebiegła bez problemu, bez konieczności zagłębiania się w szczegóły implementacji klasy `Record`.

Wzorzec projektowy *Adapter* warto stosować zawsze wtedy, gdy występują dwa interfejsy API, które muszą ze sobą współpracować, a modyfikacja żadnego z tych interfejsów nie wchodzi w rachubę.

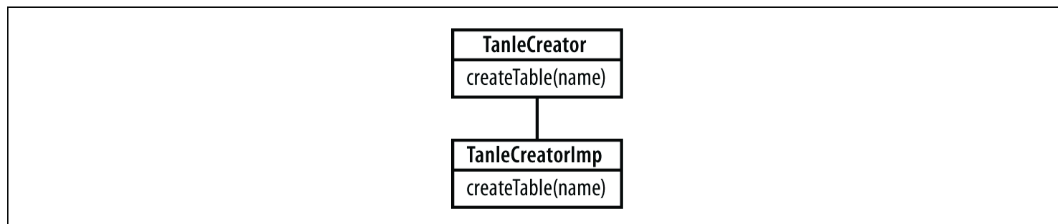
SPOSÓB 73.

Pisanie przenośnego kodu z wykorzystaniem wzorca Most

Wykorzystanie wzorca *Most* w celu ukrycia szczegółów implementacji obiektów lub modyfikacji implementacji na podstawie środowiska.

W jednej z firm, w której pracowałem, tworzyliśmy dużą aplikację w C++, która działała na wielu platformach. Podczas prac nad nią wielokrotnie wykorzystaliśmy wzorzec *Most*. Jego podstawową cechą jest możliwość ukrycia części implementacji klasy w innej klasie po to, by nie dopuścić do oglądania implementacji przez innych programistów lub dlatego, że część implementacji zależy od platformy.

W przykładzie zaprezentowanym w niniejszym podrozdziale, w celu pokazania zalet wzorca *Most*, wykorzystamy przypadek, w którym część implementacji zależy od platformy. Na rysunku 7.7 pokazano związki pomiędzy klasami `TableCreator` i `TableCreatorImp`. Rola pierwszej z nich polega na tworzeniu tabel w docelowej bazie danych. Klasę implementacyjną — `TableCreatorImp` — zdefiniowano w innym pliku, który jest włączany z katalogu specyficznego dla określonego typu bazy danych.



Rysunek 7.7. Klasa TableCreator i jej klasa implementacyjna

Dzięki takiej implementacji można stworzyć jedną wersję kodu specyficzną dla systemu Oracle i inną dla bazy MySQL (lub innej bazy danych). Jest to bardzo przydatne, zwłaszcza że w poszczególnych typach bazach danych występują różnice w składni kodu tworzącego tabele.

Kod

Kod pokazany na listingu 7.7 zapiszemy w pliku *bridge.php*.

Listing 7.7. Klasa bazowa wzorca Most

```
<?php
require( "sql.php" );

class TableCreator
{
  static function createTable( $name )
  {
    TableCreatorImp::createTable( $name );
  }
}

TableCreator::createTable( "customer" );
?>
```

Kod pokazany na listingu 7.8 zapiszemy w pliku *mysql/sql.php*.

Listing 7.8. Przykładowa klasa implementacyjna dla bazy danych MySQL

```
<?php
class TableCreatorImp
{
  static public function createTable( $name )
  {
    echo( "Wersja klasy createTable dla bazy MySQL tworząca tabelę $name\n" );
  }
}
?>
```

Kod pokazany na listingu 7.9 zapiszemy w pliku *oracle/sql.php*.

Listing 7.9. Przykładowa klasa implementacyjna dla bazy danych Oracle

```
<?php
class TableCreatorImp
{
  static public function createTable( $name )
  {
    echo( " Wersja klasy createTable dla bazy Oracle tworząca tabelę $name\n"
);
  }
}
?>
```

Wykorzystanie sposobu

Wykorzystanie zaprezentowanego sposobu wymaga zastosowania dodatkowych parametrów w wierszu polecenia informujących interpreter PHP o tym, że w ścieżce plików włączanych ma się znaleźć katalog *mysql* lub *oracle* (co oznacza użycie mostu specyficznego dla określonego typu bazy danych). Oto wersja polecenia dla bazy danych MySQL:

```
%php -d include_path = '.:/usr/local/php5/lib/php:mysql' bridge.php
Wersja klasy createTable dla bazy MySQL tworząca tabelę customer
```

A oto wersja dla bazy danych Oracle:

```
%php -d include_path = '.:/usr/local/php5/lib/php:oracle' bridge.php
Wersja klasy createTable dla bazy Oracle tworząca tabelę customer
```

Nie jest to skomplikowany przepis na zrobienie rakiety, zatem zrozumienie idei przykładu nie powinno przysporzyć trudności. Klasa `TableCreator` została zaimplementowana przez jedną z kilku wersji klasy `TableCreatorImp` umieszczonych w katalogach specyficznych dla platformy.

Oczywiście kod zamieszczony w przykładzie nie tworzy tabel. Jest to jedynie szkielet, w którym w praktycznej aplikacji trzeba by było wprowadzić odpowiedni kod. Arkana tworzenia tabel w różnych systemach baz danych nie są jednak istotne dla zrozumienia idei wzorca *Most* (można je zatem skwitować zdaniem „proszę zapoznać się z tym samodzielnie”).

Jedną z poważnych wad wzorca *Most* jest brak możliwości rozszerzania implementacji określonych klas. W tym przypadku nie stanowi to problemu, ponieważ wszystkie metody klas implementacyjnych są statyczne. Jednak w przypadku obiektów zawierających metody niestatyczne klasa implementacyjna dziedziczy cechy klas nieimplementacyjnych. Na przykład klasa `CButtonImp` dziedziczy cechy po klasie `CButton`. W celu rozszerzenia implementacji trzeba by zastosować dziedziczenie po klasie `CButtonImp`, która jest ukryta. Problem ten dotyczy jednak w większym stopniu języków kompilowanych, takich jak C++.

SPOSÓB
74.

Rozszerzalne przetwarzanie z wykorzystaniem wzorca Łańcuch odpowiedzialności

Wykorzystanie wzorca *Łańcuch odpowiedzialności* do utworzenia szkieletu kodu w trybie Plug and Play.

Oglądanie futbolu z programistami jest zabawne. Nawet w czwartej kwarcie, kiedy wynik meczu wynosi 33:7, a pozostało zaledwie półtorej minuty do końca, w dalszym ciągu wskazują na wiele możliwości ostatecznego wyniku. Jest tak dlatego, że są przyzwyczajeni do przewidywania wszystkich sytuacji niezależnie od tego, jak bardzo są nieprawdopodobne (a właściwie zupełnie absurdalne). Przekonałem się, że większość programistów, włącznie ze mną, nie znosi zamykania drzwi odnośnie odpowiedzi na żadne z pytań. Lepiej napisać kod obsługujący 100 możliwych przypadków nawet wtedy, gdy nasz menedżer zaklina się, że jest tylko jedna możliwość.

Dlatego właśnie wzorzec projektowy *Łańcuch odpowiedzialności* (ang. *Chain of responsibility*) jest tak ważny. Wyobraźmy sobie, że do pomieszczenia, w którym jest wiele osób, wchodzi sprzedawca ciastek, niosąc karton z pączkami o różnych smakach. Otwiera torebkę i wyjmuje pączek z marmoladą. Po kolei pyta poszczególne osoby, czy życzą sobie pączka z marmoladą, aż znajdzie się ktoś, kto będzie chciał. Następnie powtarza czynność dla pozostałych pączków z torebki do czasu, aż będzie pusta.

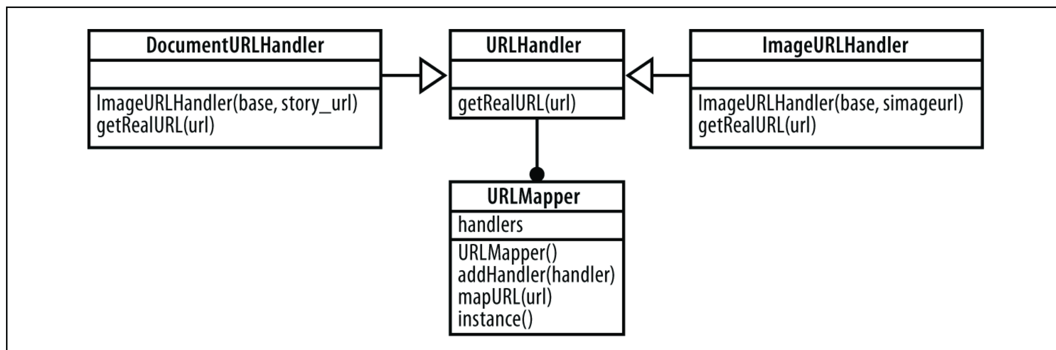
To właśnie jest łańcuch odpowiedzialności. Każda osoba w pokoju rejestruje się wcześniej u dostawcy pączków. Kiedy przychodzi nowa partia pączków, dostawca widzi, kto je zamawiał, patrząc na listę zarejestrowanych osób. Zaleta tej sytuacji polega na tym, że dostawcy pączków nie interesuje, ile osób zamawia pączki, nie interesuje go nawet, co z nimi zrobią. Zajmuje się tylko zarządzaniem rejestracją i dostawami.

W podrozdziale napiszę kod, w którym zamiast pączków będę posługiwał się adresami URL. Skrypt będzie dostarczał adresy URL do kilku procedur obsługi, które potencjalnie będą je przekierowywały. Jeśli żadna z procedur obsługi nie obsłuży adresu URL, taki adres będzie zignorowany.

Na rysunku 7.8 pokazano, w jaki sposób ma działać ten system. Klasa `URLMapper` to dostawca pączków. Ma karton pełen adresów URL, które zamierza wręczyć obiektom o interfejsie `URLHandler`, które się po nie zgłoszą. W tym przypadku klasa `ImageURLHandler` zarządza kierowaniem żądań adresów URL plików graficznych do skryptu obsługującego grafikę. W podobny sposób obiekt `DocumentURLHandler` przekierowuje żądania dokumentów do odpowiednich stron PHP. Dzięki temu aplikacja może przesłać adresy URL bez specjalnego kodu obsługi, a jednocześnie modyfikować je w miarę potrzeb.

Kod

Kod pokazany na listingu 7.10 zapiszemy w pliku *chain.php*.



Rysunek 7.8. Interfejs URLHandler, obiekt przekierowujący i dwa obiekty obsługujące adresy URL

Listing 7.10. Przykład zastosowania w PHP wzorca projektowego Łańcuch odpowiedzialności

```
<?php
abstract class URLHandler
{
  abstract function getRealURL( $url );
}

class URLMapper
{
  private $handlers = array();

  private function URLMapper()
  {
  }

  public function addHandler( $handler )
  {
    $this->handlers []= $handler;
  }

  public function mapURL( $url )
  {
    foreach( $this->handlers as $h )
    {
      $mapped = $h->getRealURL( $url );
      if ( isset( $mapped ) ) return $mapped;
    }
    return $url;
  }

  public static function instance()
  {
    static $inst = null;
    if( !isset( $inst ) ) { $inst = new URLMapper(); }
    return $inst;
  }
}

class ImageURLHandler extends URLHandler
{
  private $base;
  private $imgurl;

  public function ImageURLHandler( $base, $imgurl )
```

```
  {
    $this->base = $base;
    $this->imgurl = $imgurl;
  }

  public function getRealURL( $url )
  {
    if ( preg_match( "|^".$this->base."(.*?)$|", $url, $matches ) )
    {
      return $this->imgurl.$matches[1];
    }
    return null;
  }
}

class DocumentURLHandler extends URLHandler
{
  private $base;
  private $story_url;

  public function DocumentURLHandler( $base, $story_url )
  {
    $this->base = $base;
    $this->story_url = $story_url;
  }

  public function getRealURL( $url )
  {
    if ( preg_match( "|^".$this->base."(.*?)/(.*?)/(.*?)$|", $url, $matches )
)
    {
      return $this->story_url.$matches[1].$matches[2].$matches[3];
    }
    return null;
  }
}


$ih = new ImageURLHandler( "http://mysite.com/images/",
"http://mysite.com/image.php?img=" );
URLMapper::instance()->addHandler( $ih );
$ih = new DocumentURLHandler( "http://mysite.com/story/",
"http://mysite.com/story.php?id=" );
URLMapper::instance()->addHandler( $ih );

$testurls = array();
$testurls []= "http://mysite.com/index.html";
$testurls []= "http://mysite.com/images/dog";
$testurls []= "http://mysite.com/story/11/05/05";
$testurls []= "http://mysite.com/images/cat";
$testurls []= "http://mysite.com/image.php?img=lizard";

foreach( $testurls as $in )
{
  $out = URLMapper::instance()->mapURL( $in );
  print "$in\n -->  $out\n\n";
}
?>
```

Wykorzystanie sposobu

Skrypt *chain.php* uruchomimy za pomocą interpretera PHP działającego w wierszu polecenia:

```
%php chain.php
http://mysite.com/index.html
 -->  http://mysite.com/index.html

http://mysite.com/images/dog
 -->  http://mysite.com/image.php?img=dog

http://mysite.com/story/11/05/05
 -->  http://mysite.com/story.php?id=110505

http://mysite.com/images/cat
 -->  http://mysite.com/image.php?img=cat

http://mysite.com/image.php?img=lizard
 -->  http://mysite.com/image.php?img=lizard
%
```

Każdy wchodzący adres URL jest przesyłany poprzez obiekt `URLMapper`, który zwraca adres po jego przekształceniu. Pierwszy adres URL nie jest przekierowywany, zatem obiekt `URLMapper` przekazuje go w niezmienionej postaci. W drugim przypadku obiekt `ImageURLHandler` wykrywa, że adres URL dotyczy grafiki, zatem kieruje go do skryptu *image.php*. Trzeci adres został rozpoznany jako dokument, zatem skierowano go do skryptu *story.php*.

Doskonałą własnością wzorca projektowego *Łańcuch odpowiedzialności* jest możliwość jego rozszerzania bez konieczności modyfikacji kodu aplikacji. Wystarczy, że obiekt dostawcy będzie wyposażony w dostatecznie rozbudowany interfejs API dla zarejestrowanych obiektów, aby obsłużyć niemal każdą sytuację.

Jednym z najbardziej rozpoznawanych przykładów wzorca *Łańcuch odpowiedzialności* jest serwer WWW Apache, który działa jak jeden wielki dostawca pączków, delegując rożne żądania do zarejestrowanych procedur obsługi.

Wzorzec *Łańcuch odpowiedzialności* nie zawsze jest łatwy do zastosowania. Jest z nim związanych kilka poważnych problemów. Trudno poprawia się w nim błędy i nie zawsze wiadomo, w jaki sposób należy go właściwie wykorzystywać. Występują dwie odmiany wzorca: jedna, w której w przypadku znalezienia procedury obsługi żądanie nie jest dalej przesyłane, i druga, gdzie przetwarzanie jest kontynuowane niezależnie od tego, czy znaleziono właściwą procedurę obsługi. Nie zawsze wiadomo, która z wersji jest wykorzystywana. Co więcej, drugi wariant, gdzie zdarzają się sytuacje wywołania wielu procedur obsługi, jest szczególnie trudny do diagnozowania. W przypadku wzorca *Łańcuch odpowiedzialności* potwierdza się reguła, że rozbudowane możliwości programów komputerowych osiąga się kosztem złożoności i wydajności.

SPOSÓB 75.

Podział rozbudowanych klas na mniejsze z wykorzystaniem wzorca Kompozyt

Wykorzystanie wzorca *Kompozyt* w celu podzielenia rozbudowanych klas na mniejsze.

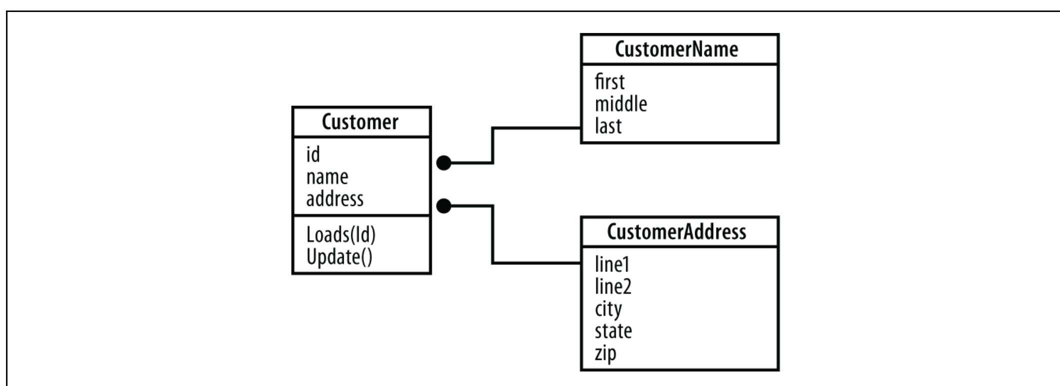
Kiedy słyszę informacje o wielkich bazach danych, w których są zapisane wszelkie informacje o osobach, których ktokolwiek i kiedykolwiek mógłby potrzebować, reaguję bardzo dziwnie. Większość osób myśli pewnie o prywatności, mnie przychodzi do głowy myśl o tym, jak źle zaprojektowano taki system. Jestem niemal pewien, że jest w nim jeden megaobiekt `Person` zawierający jakieś 4 000 pól i 8 000 metod.

Skąd to wiem? Ponieważ sam miałem do czynienia z takimi obiektami! Dla takiej klasy koniecznie trzeba zastosować wzorzec *Kompozyt*. Dzięki niemu klasa `Person` pozostanie, ale owe 4 000 pól zostanie podzielone na obiekty pochodne. W obiekcie klasy `Person` pozostanie około 100 obiektów, z których każdy będzie zawierał inne, mniejsze obiekty (potencjalnie zawierające jeszcze mniejsze obiekty itd.).

Nie chcę powiedzieć, że zetknąłem się z tak źle zaprojektowanymi klasami w PHP, choć spotykałem klasy, w których było ponad 100 pól tylko dlatego, że obiekty reprezentowały zbiór tabel zawierających wiele pól związanych z jednym zapisem. W podrozdziale pokazałem sposób podziału klasy `Customer` (w której jest o wiele za dużo pól) na kilka mniejszych klas. Na końcu procesu pozostaje jedna, złożona klasa `Customer`.

W pokazanym przykładzie podzieliłem klasę zawierającą około ośmiu pól. Listing ten można ekstrapolować dla klas zawierających kilkaset takich pól, z jakimi spotykałem się wcześniej.

Budowę klasy `Customer` pokazano na rysunku 7.9. Zawiera ona po jednym obiekcie `CustomerName` i `CustomerAddress`.



Rysunek 7.9. Złożona klasa Customer wraz z jej klasami potomnymi

Kod

Kod pokazany na listingu 7.11 zapiszemy w pliku *composite.php*.

Listing 7.11. Obiekt Customer złożony z mniejszych obiektów

```
<?php
class CustomerName
{
  public $first = "";
  public $middle = "";
  public $last = "";
}

class CustomerAddress
{
  public $line1 = "";
  public $line2 = "";
  public $city = "";
  public $state = "";
  public $zip = "";
}

class Customer
{
  public $id = null;
  public $name = null;
  public $address = null;

  public function Customer()
  {
    $this->name = new CustomerName();
    $this->address = new CustomerAddress();
  }

  public function Load( $id )
  {
    $this->id = $id;
    $this->name->first = "George";
    $this->name->middle = "W";
    $this->name->last = "Bush";
    $this->address->line1 = "1600 Pennsylvania Ave.";
    $this->address->line2 = "";
    $this->address->city = "Washington";
    $this->address->state = "DC";
    $this->address->zip = "20006";
  }

  public function Update()
  {
    // Aktualizacja rekordu w bazie danych
    // lub wprowadzenie rekordu, jeśli nie ma identyfikatora.
  }

  public function __toString()
  {
    return $this->name->first." ".$this->name->last;
  }
}
```

```
$cust = new Customer();
$cust->Load( 1 );
print( $cust );
print( "\n" );
?>
```

Wykorzystanie sposobu

Powyższy skrypt uruchomimy, wykorzystując interpreter PHP działający w wierszu polecenia:

```
%php composite.php
George Bush
```

Skrypt tworzy nowego klienta i ładuje rekord numer 1. W przykładzie zakodowałem „na sztywno” dane George W. Busha. Następnie skrypt wyświetla informacje zapisane w obiekcie `Customer`.

Nie ma w tym nic wielkiego. Idea przykładu jest równie skuteczna jak prosta. Nie należy używać megaklas zawierających po 100 pól. O wiele lepiej jest posługiwać się niewielkimi pogrupowanymi klasami, takimi jak `CustomerName` i `CustomerAddress`, które można wkomponować w większe struktury — w tym przypadku klasę `Customer`. Co więcej, klasę `CustomerAddress` można wykorzystać w innych klasach, gdzie istnieje potrzeba wykorzystania adresów pocztowych.

Wzorzec *Kompozyt* warto zastosować w przypadku, gdy dane obiektu są rozproszone w wielu tabelach bazy danych. Każdej z tabel powinna odpowiadać własna klasa lub inna struktura danych.

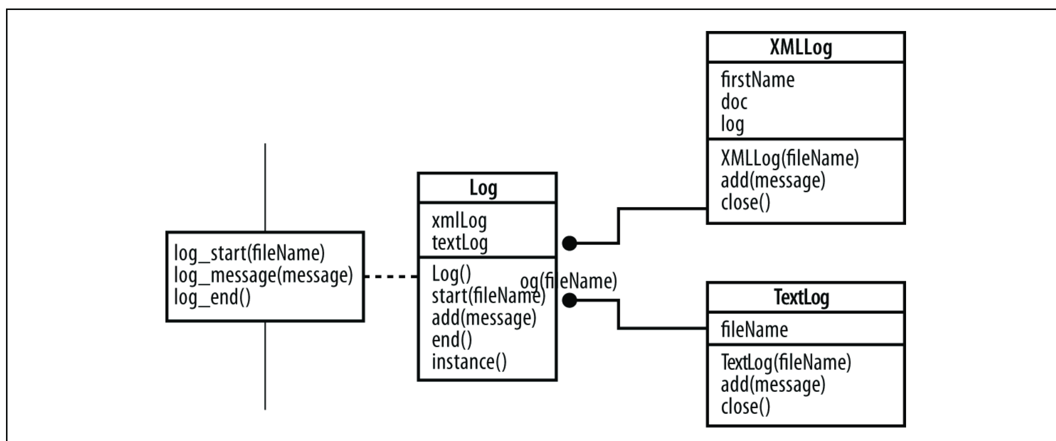
Wzorzec *Kompozyt* ułatwia optymalizację odczytu informacji z bazy danych. Ponieważ załadowanie każdego z obiektów podrzędnych, takich jak adres, wymaga osobnego zapytania, dane można odczytywać w niewielkich porcjach. Inaczej mówiąc, można opóźnić ładowanie określonego podobiektu do czasu, kiedy zapisane w nim dane będą potrzebne. Dzięki temu kod nie musi pobierać setek pól z wielu tabel w przypadku, kiedy potrzebne jest jedynie imię i nazwisko klienta.

SPOSÓB 76. Uproszczenie interfejsu API z wykorzystaniem wzorca Fasada

Wykorzystanie wzorca *Fasada* w celu uproszczenia interfejsu API prezentowanego innym programistom.

Wzorzec *Fasada* jest jednym z tych, które moim zdaniem powinno stosować więcej programistów, i to nie z powodu ładnie brzmiącej nazwy, ale dlatego, że jeśli ktoś stosuje wzorzec *Fasada*, to znaczy, że myśli o innych programistach i o tym, by uzyskali właśnie te informacje, których potrzebują (i nic więcej, dzięki czemu nie mogą nic zepsuć).

Weźmy za przykład prosty interfejs API systemu rejestrowania przedstawiony na rysunku 7.10.



Rysunek 7.10. Interfejs API systemu rejestrowania z prostym przykładem wzorca Fasada

Pokazany interfejs API umożliwia rejestrowanie zdarzeń w formacie XML, tekstowym lub obu. Jako programista jestem pod wrażeniem umiejętności autora. Wydaje się, że są dostępne metody dla wszystkich informacji: rozpoczęcia komunikatu, wprowadzenia tekstu, porządkowania, a nawet obsługi formatu XML i tekstowego.

Naprawdę jednak interesuje mnie, jakich metod mam używać i kiedy. Właśnie do tego służy wzorzec *Fasada* — jego zastosowanie daje pewność prawidłowego wykorzystywania interfejsu API. Wzorzec *Fasada* zastosowany w tym przykładzie to lista trzech funkcji wyświetlanych w ramce, przez którą przechodzi linia po lewej stronie rysunku. Ta linia to rodzaj teoretycznej bariery, na której wyświetla się informacja: „jestem odpowiedzialny za operacje zdefiniowane po prawej stronie; wywołuj moje metody, a ja zajmę się resztą".

Zastosowanie wzorca *Fasada* nie tylko upraszcza interfejsy API, ale także ukrywa szczegóły implementacji przed klientami. Implementacja może się zmienić, a klient nawet tego nie zauważy. Jest to równie ważne jak uproszczenie interfejsów. Należy pamiętać, że luźne wiązanie obiektów oznacza stabilne i niezawodne systemy.

Kod

Kod pokazany na listingu 7.12 zapiszemy w pliku *test.php*.

Listing 7.12. Kod testowy systemu rejestrowania zdarzeń

```
<?php
require( "log.php" );

log_start( "mylog" );
log_message( "Otwarcie aplikacji" );
log_message( "Zarejestrowanie komunikatu" );
log_message( "Zamknięcie aplikacji" );
log_end();
?>
```

Kod pokazany na listingu 7.13 zapiszemy w pliku *log.php*.

Listing 7.13. Zastosowanie wzorca Fasada

```
<?php
require( "log_impl.php" );

function log_start( $fileName )
{
  Log::instance()->start( $fileName );
}

function log_message( $message )
{
  Log::instance()->add( $message );
}

function log_end()
{
  Log::instance()->end();
}
?>
```

Kod pokazany na listingu 7.14 zapiszemy w pliku *log_impl.php*.

Listing 7.14. Implementacja wzorca Fasada

```
<?php
class XMLLog
{
  private $fileName;
  private $doc;
  private $log;

  public function XMLLog( $fileName )
  {
    $this->fileName = $fileName;

    $this->doc = new DOMDocument();
    $this->doc->formatOutput = true;
    $this->log = $this->doc->createElement( "log" );
    $this->doc->appendChild( $this->log );
  }

  public function add( $message )
  {
    $mess_obj = $this->doc->createElement( "message" );
    $text = $this->doc->createTextNode( $message );
    $mess_obj->appendChild( $text );
    $this->log->appendChild( $mess_obj );
  }

  public function close()
  {
    $this->doc->save( $this->fileName );
  }
}

class TextLog
{
  private $fh;
```

```
  public function TextLog( $fileName )
  {
    $this->fh = fopen( $fileName, "w" );
  }

  public function add( $message )
  {
    fprintf( $this->fh, $message."\n" );
  }

  public function close()
  {
    fclose( $this->fh );
  }
}

class Log
{
  private $xmlLog = null;
  private $textLog = null;

  public function Log()
  {
  }

  public function start( $fileName )
  {
    $this->xmlLog = new XMLLog( $fileName.".xml" );
    $this->textLog = new TextLog( $fileName.".txt" );
  }

  public function add( $message )
  {
    $this->xmlLog->add( $message );
    $this->textLog->add( $message );
  }

  public function end()
  {
    $this->xmlLog->close();
    $this->textLog->close();
  }

  public static function instance()
  {
    static $inst = null;
    if ( !isset( $inst ) ) $inst = new Log();
    return $inst;
  }
}
?>
```

Wykorzystanie sposobu

Zaprezentowany kod uruchomimy za pomocą interpretera PHP działającego w wierszu polecenia:

```
% php test.php
% cat mylog.txt
Otwarcie aplikacji
Zarejestrowanie komunikatu
```

```
Zamknięcie aplikacji
% cat mylog.xml
<?xml version="1.0"?>
<log>
  <message>Otwarcie aplikacji</message>
  <message>Zarejestrowanie komunikatu</message>
  <message>Zamkni&#x119;cie aplikacji</message>
</log>
```

Nie ma specjalnie czego oglądać, ale w rzeczywistości interesuje nas kod (a nie jego wynik). W skrypcie *test.php* następuje rozpoczęcie pliku dziennika, wysłanie kilku komunikatów, a następnie jego zamknięcie. Operacje te są wykonywane za pomocą zaledwie trzech funkcji zdefiniowanych w skrypcie z wzorcem *Fasada* — *log.php*. W idealnym środowisku *log.php* byłby jedynym skryptem, do którego mieliby dostęp programiści „z zewnątrz".

W skrypcie *log.php* do utworzenia dwóch dzienników zastosowano obiekt `Log` zaimplementowany za pomocą wzorca *Singleton* [Sposób 77.] w pliku *log_impl.php*. Skrypt *log.php* wysyła po jednym komunikacie do każdego z dzienników, a następnie umieszcza je w odpowiednich plikach (tekstowym lub XML).

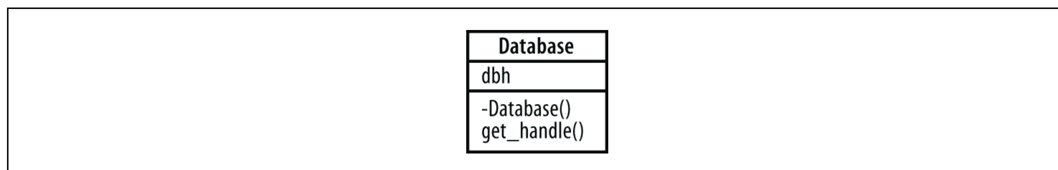
SPOSÓB 77. Tworzenie stałych obiektów za pomocą wzorca Singleton

Wykorzystanie wzorca *Singleton* do utworzenia obiektów, które występują w systemie jako pojedyncze egzemplarze.

Spośród wszystkich wzorców projektowych opisanych w książce *Design Patterns* autorstwa „Gangu czterech" żaden nie jest wykorzystywany tak często jak *Singleton*. Częściowo przyczyną tego faktu jest jego łatwa implementacja. Zresztą, czy może być coś lepszego od zakodowania obiektu typu *Singleton* i dumnego oświadczenia, że taki obiekt może być tylko jeden? Jest w tym coś z *Nieśmiertelnego*.

Singleton to typ obiektowy, dla którego w określonym momencie może w systemie występować tylko jeden egzemplarz. Wzorzec ten doskonale nadaje się do implementacji uchwytu do bazy danych. Dla każdego egzemplarza interpretera PHP może występować tylko jeden uchwyt do bazy danych. Właśnie taką konfigurację zaprezentuję w tym podrozdziale.

Diagram UML uchwytu do bazy danych z wykorzystaniem wzorca *Singleton* pokazano na rysunku 7.11 (prawda, że proste?).



Rysunek 7.11. Obiekt Singleton dostępu do bazy danych

Rzeczywiście nie ma na co patrzeć. Obiekt zawiera uchwyt do bazy danych oraz dwie metody. Pierwsza to konstruktor, który jest prywatny po to, by mieć pewność, że kod spoza klasy nie będzie w stanie utworzyć obiektu. Druga to statyczna metoda `get_handle` zwracająca uchwyt do bazy danych.

Kod

Kod pokazany na listingu 7.15 zapiszemy w pliku *singleton1.php*.

Listing 7.15. Zastosowanie wzorca singleton jako klasy opakowującej dla bazy danych

```
<?php
require( 'DB.php' );

class Database
{
  private $dbh;

  private function Database()
  {
    $dsn = 'mysql://root:password@localhost/test';
    $this->dbh =& DB::Connect( $dsn, array() );
    if (PEAR::isError($this->dbh)) { die($this->dbh->getMessage()); }
  }

  public static function get_handle()
  {
    static $db = null;
    if ( !isset($db) ) $db = new Database();
    return $db->dbh;
  }
}

echo( Database::get_handle()."\n" );
echo( Database::get_handle()."\n" );
echo( Database::get_handle()."\n" );
?>
```

Ten prosty obiekt zgodny z wzorcem *Singleton* zawiera konstruktor obsługujący logowanie do bazy danych oraz jedną statyczną metodę dostępową, która tworzy obiekt, jeśli nie został utworzony wcześniej, i zwraca odczytany z niego uchwyt do bazy danych. Skorzystanie z tej metody w celu odczytania uchwytu do bazy danych daje pewność, że połączenie z bazą danych uzyskamy tylko raz podczas danego żądania strony.

Wykorzystanie sposobu

Skrypt uruchomimy, korzystając z interpretera PHP działającego w wierszu polecenia:

```
%php singleton1.php
Object id#2
Object id#2
Object id#2
```

Wykonanie przykładu dowodzi tego, że wiele wywołań statycznej metody `get_handle()` za każdym razem zwraca ten sam obiekt, a tym samym zapewnia skorzystanie z tego samego uchwytu do bazy danych.

Modyfikacja sposobu

Z uchwytami do bazy danych poszło łatwo. Zastanówmy się jednak, czy można wykorzystać wzorzec *Singleton* dla bardziej skomplikowanych obiektów? Spróbujmy użyć go dla współdzielonej listy stanów, tak jak pokazano na listingu 7.16.

Listing 7.16. Tablica stanów zaimplementowana za pomocą wzorca Singleton

```
<?php
class StateList
{
  private $states = array();

  private function StateList()
  {
  }

  public function addState( $state )
  {
    $this->states []= $state;
  }

  public function getStates()
  {
    return $this->states;
  }

  public static function instance()
  {
    static $states = null;
    if ( !isset($states) ) $states = new StateList();
    return $states;
  }
}

StateList::instance()->addState( "Florida" );
var_dump( StateList::instance()->getStates() );

StateList::instance()->addState( "Kentucky" );
var_dump( StateList::instance()->getStates() );
?>
```

Powyższy kod tworzy klasę `StateList` zawierającą listę stanów. Do listy można dodawać stany, a także odczytać stany, które są już na liście. Do uzyskania pojedynczego, współdzielonego egzemplarza tego obiektu trzeba skorzystać ze statycznej metody `instance()` (zamiast bezpośredniego tworzenia egzemplarza).

Do uruchomienia skryptu wykorzystamy interpreter PHP działający w wierszu polecenia:

```
% php singleton2.php
array(1) {
    [0]=>
    string(7) "Florida"
}
array(2) {
    [0]=>
    string(7) "Florida"
    [1]=>
    string(8) "Kentucky"
}
```

Z pierwszego zrzutu widać, że na liście znajduje się pierwszy stan — Floryda. Drugi zrzut dowodzi, że do listy współdzielonego obiektu dodano drugi stan — Kentucky.

Jeśli mam być szczery, nie polecam zbyt częstego wykorzystywania wzorca *Singleton*. Według mnie jest on wykorzystywany zbyt często. Niejednokrotnie miałem do czynienia z kodem, gdzie stosowano pewne niezgrabne obejścia w celu wykorzystania obiektów *Singleton*. Bardzo często oznaczało to niepoprawne korzystanie z wzorca *Singleton*. Jeśli zastosowanie wzorca *Singleton* wymaga zbyt wielu operacji, może to oznaczać, że wzorzec ten nie został zastosowany we właściwym miejscu.

SPOSÓB 78.

Ułatwienie wykonywania operacji z danymi dzięki zastosowaniu wzorca Wizytator

Wykorzystanie wzorca *Wizytator* do oddzielenia przeglądania danych od ich przetwarzania.

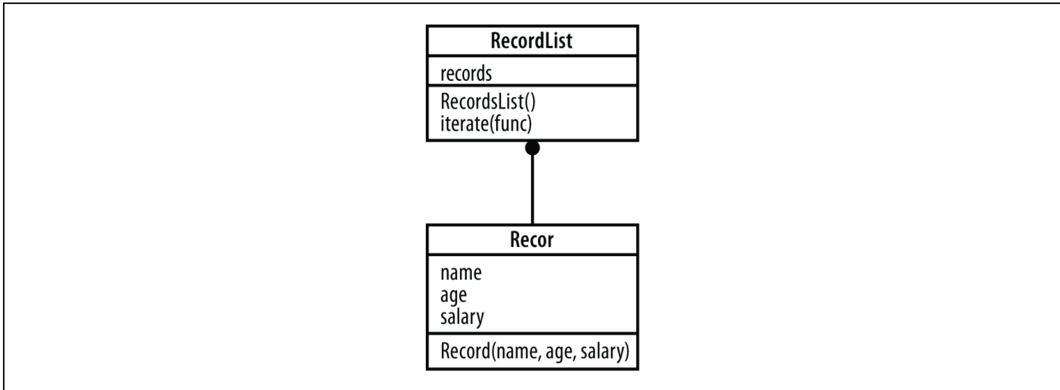
Na początku mojej kariery programistycznej tworzyłem wiele programów wykonujących obliczenia naukowe, w których wykorzystywano systemy zbierania danych. Były to systemy rejestrujące próbki danych w odstępach co 3 mikrosekundy — inaczej mówiąc, 333 333 próbek na sekundę. Taka częstotliwość pobierania informacji oznaczała 38 megabajtów danych na minutę! W przypadku długo trwających sesji rozmiar pliku z danymi z łatwością przekraczał kilka gigabajtów. Nie trzeba dodawać, że rejestrowanie takich ilości informacji i zapisywanie ich na dysku bez zastosowania specjalnych chwytów sprawiało kłopoty.

Osobnym problemem było analizowanie tych danych. W jaki sposób analizować plik o rozmiarze kilku gigabajtów, jeśli komputer, którego używamy do tego celu, ma zaledwie 128 MB pamięci? Wiadomo, że trzeba podzielić dane. Oznacza to odczytywanie pliku sekcja po sekcji, wyrzucanie niepotrzebnych danych z pamięci na dysk i wczytywanie tych, które są potrzebne, z dysku do pamięci.

Algorytmy stosowane we wspomnianych programach naukowych były i tak dostatecznie rozbudowane, nawet bez obsługi wymiany danych z dyskiem, a co dopiero z nią. Aby elegancko rozwiązać nasze problemy, zastosowaliśmy wzorzec *Wizytator* (ang. *Visitor*). Jeden obiekt był odpowiedzialny za wymianę danych pomiędzy pamięcią a dyskiem, a drugi za przetwarzanie ich w pamięci.

Na rysunku 7.12 pokazano obiekt `RecordList` zawierający listę obiektów `Record`. Jest w nim metoda `iterate()`, która pobiera argument w postaci nazwy innej funkcji i wywołuje ją dla każdego rekordu.

Dzięki takiemu podejściu funkcja przetwarzania danych przekazywana do metody `iterate()` nie musi znać sposobu zarządzania rekordami w pamięci. Jedyne działania, jakie musi wykonywać, to obsługiwać przekazane do niej dane.



Rysunek 7.12. Obiekt RecordList z metodą iterate

Kod

Kod zaprezentowany na listingu 7.17 zapiszemy w pliku *visitor1.php*.

Listing 7.17. Zastosowanie wzorca Wizytator do przeglądania rekordów w bazie danych

```
<?php
class Record
{
  public $name;
  public $age;
  public $salary;
  public function Record( $name, $age, $salary )
  {
    $this->name = $name;
    $this->age = $age;
    $this->salary = $salary;
  }
}

class RecordList
{
  private $records = array();

  public function RecordList()
  {
    $this->records []= new Record( "Leszek", 22, 35000 );
    $this->records []= new Record( "Henryk", 25, 37000 );
    $this->records []= new Record( "Maria", 42, 65000 );
    $this->records []= new Record( "Stefania", 45, 80000 );
  }

  public function iterate( $func )
  {
    foreach( $this->records as $r )
    {
      call_user_func( $func, $r );
    }
  }
}

$min = 100000;
```

```
function find_min_salary( $rec )
{
  global $min;
  if( $rec->salary < $min ) { $min = $rec->salary; }
}

$rl = new RecordList();
$rl->iterate( "find_min_salary", $min );
echo( $min."\n" );
?>
```

Wykorzystanie sposobu

Do uruchomienia skryptu zaprezentowanego powyżej wykorzystamy interpreter PHP działający w wierszu polecenia:

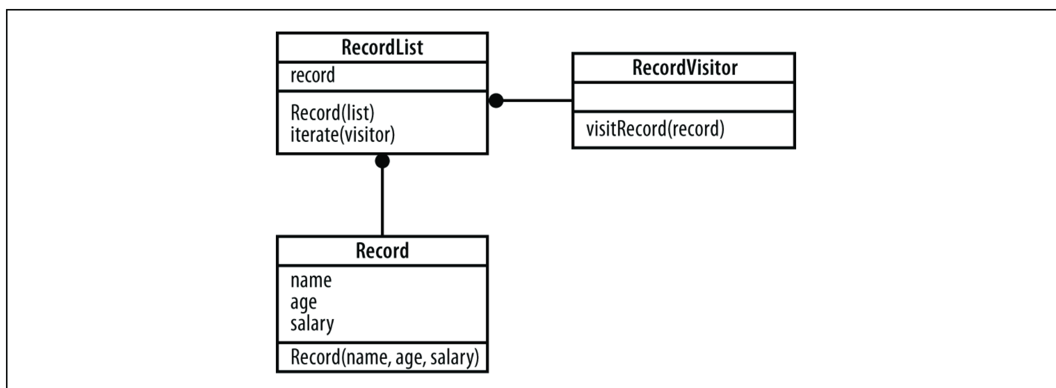
```
% php visitor1.php
35000
```

Zaprezentowany algorytm wybiera rekord osoby o najniższej pensji spośród wszystkich przetwarzanych rekordów. Kod skryptu jest stosunkowo prosty. Klasa `Record` zawiera dane poszczególnych rekordów. Klasa `RecordList` ładuje się z pewnymi przykładowymi danymi (w praktycznym zastosowaniu dane te można by odczytać z bazy danych lub pliku). Metoda `iterate()` w pętli **`foreach`**`()` przetwarza listę rekordów. Metoda `call_user_func()` wywołuje przekazaną do niej funkcję przetwarzającą dane dla każdego rekordu. W tym przykładzie jest to funkcja `find_min_salary()`, która przegląda poszczególne rekordy w celu znalezienia najniższej wartości pensji.

Modyfikacja sposobu

Wersja zastosowania wzorca *Wizytator* z funkcjami jest według mnie trochę niezgrabna. Lepiej byłoby zdefiniować obiekt-wizytator odczytujący poszczególne rekordy. Dzięki temu dane o wartości minimalnej mogłyby być zapisane w obiekcie i odczytane w późniejszym czasie.

Na rysunku 7.13 pokazano odmianę implementacji wzorca *Wizytator*, gdzie obiekt typu `RecordVisitor` pobiera metoda `iterate()`, a nie funkcja.



Rysunek 7.13. Implementacja wzorca Wizytator, w której wizytator jest obiektem

Zaktualizowany kod zaprezentowano na listingu 7.18.

Listing 7.18. Zaktualizowana wersja implementacji wzorca Wizytator

```
<?php
class Record
{
  public $name;
  public $age;
  public $salary;
  public function Record( $name, $age, $salary )
  {
    $this->name = $name;
    $this->age = $age;
    $this->salary = $salary;
  }
}

abstract class RecordVisitor
{
  abstract function visitRecord( $rec );
}

class RecordList
{
  private $records = array();

  public function RecordList()
  {
    $this->records []= new Record( "Leszek", 22, 35000 );
    $this->records []= new Record( "Henryk", 25, 37000 );
    $this->records []= new Record( "Maria", 42, 65000 );
    $this->records []= new Record( "Stefania", 45, 80000 );
  }

  public function iterate( $vis )
  {
    foreach( $this->records as $r )
    {
        $vis->visitRecord( $r );
    }
  }
}

class MinSalaryFinder extends RecordVisitor
{
  public $min = 1000000;
  public function visitRecord( $rec )
  {
    if( $rec->salary < $this->min ) { $this->min = $rec->salary; }
  }
}

$rl = new RecordList();
$msl = new MinSalaryFinder();
$rl->iterate( $msl );
echo( $msl->min."\n" );
?>
```

W tej wersji dodałem klasę abstrakcyjną `RecordVisitor` i zaimplementowałem ją za pomocą klasy `MinSalaryFinder`, która zapisuje minimalną wartość pensji. Kod testowy tworzy obiekt `RecordList`, następnie obiekt `MinSalaryFinder` i przetwarza dane z listy za pomocą metody `iterate()`. Na koniec wyświetla znalezioną wartość minimalną.

Na zakończenie warto wyciągnąć kilka wniosków dotyczących zaprezentowanego sposobu. Po pierwsze, język PHP nie najlepiej nadaje się do dynamicznego wywoływania funkcji. Specyfikowanie funkcji za pomocą nazwy jest niezręczne i stwarza dużo okazji do popełnienia błędów. W językach Python, Perl, Ruby, Java i C# (a także większości innych języków) są możliwości przypisywania wskaźnika funkcji do zmiennej. W takim przypadku można za pośrednictwem wskaźnika na funkcję wywołać metodę. Lubię PHP tak jak wielu innych programistów, ale uważam, że ten problem należałoby rozwiązać w kolejnej wersji języka.