helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

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

PHP5. Zaawansowane programowanie

Autorzy: Edward Lecky-Thompson, Heow Eide-Goodman, Steven D. Nowicki, Alec Cove
Tłumaczenie: Adam Byrtek, Jarosław Dobrzański, Paweł Gonera
ISBN: 83-7361-825-2
Tytuł oryginału: Professional PHP5
Format: B5, stron: 664

PHP to język umożliwiający tworzenie aplikacji sieciowych uruchamianych po stronie serwera. Jego najnowsza wersja, oznaczona numerem 5, to w pełni obiektowy język, za pomocą którego można budować nawet najbardziej złożone systemy portalowe, intranetowe i ekstranetowe. Dzięki nowym funkcjom wprowadzonym w PHP 5 możliwe jest korzystanie z plików XML i protokołu SOAP, wydajna komunikacja z bazą danych i stosowanie technik obiektowych znacznie ułatwiających i przyspieszających tworzenie rozbudowanych aplikacji.

„PHP5. Zaawansowane programowanie" to książka przedstawiająca potężne możliwości i elastyczność najnowszej wersji tej popularnej platformy programistycznej. Opisuje podstawowe zasady programowania obiektowego i prowadzenia rozbudowanych projektów informatycznych. Zawiera informacje o modelowaniu aplikacji w języku UML, stosowaniu wzorców projektowych i tworzeniu narzędzi, które będzie można wykorzystać w każdym projekcie. Przedstawia również analizę prawdziwego przypadku – systemu automatyzującego pracę działu sprzedaży w przedsiębiorstwie handlowym.

- Programowanie obiektowe
- Język UML i modelowanie systemów
- Wzorce projektowe
- Tworzenie prostych narzędzi obiektowych
- Połączenia z bazami danych
- Model MVC
- Stosowanie protokołu SOAP
- Komunikacja z użytkownikiem i mechanizmy sesji
- Testowanie aplikacji
- Studium przypadku – automatyzacja pracy działu sprzedaży

Jeśli chcesz poznać zaawansowane możliwości PHP 5, sięgnij po tą książkę

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

Spis treści

O Autorach 13

Wstęp 15

Część I Technologia obiektowa 21

Rozdział 1. Wprowadzenie do programowania obiektowego 23

Czym jest programowanie obiektowe? 23
Zalety programowania obiektowego 24
Przykład z życia 25
Pojęcia związane z programowaniem obiektowym 25
Klasy 26
Obiekty 27
Dziedziczenie 36
Interfejsy 45
Hermetyzacja 48
Zmiany w PHP5 dotyczące programowania obiektowego 49
Podsumowanie 50

Rozdział 2. Unified Modeling Language (UML) 51

Zbieranie wymagań 51
Rozmowa z klientem 52
Diagramy przypadków użycia 53
Diagramy klas 54
Modelowanie dziedziny 55
Relacje 56
Implementacja 58
Diagramy aktywności 61
Diagramy przebiegu 63
Diagramy stanów 65
Diagram komponentów i instalacji 66
Podsumowanie 67

Rozdział 3. Obiekty zaczynają działać ... 69
Tworzenie menedżera kontaktów ... 69
Diagramy UML dla menedżera kontaktów ... 70
Klasa PropertyObject ... 74
Klasy z informacjami kontaktowymi ... 76
Klasa DataManager ... 80
Klasy Entity, Individual i Organization ... 81
Użycie systemu ... 88
Podsumowanie ... 90

Rozdział 4. Wzorce projektowe ... 91
Wzorzec złożony ... 92
Implementacja ... 93
Przemyślenia ... 97
Wzorzec obserwatora ... 98
Widget ... 98
Przemyślenia ... 104
Wzorzec dekoratora ... 104
Implementacja ... 106
Korzystanie z dekoratora ... 107
Przemyślenia ... 108
Wzorzec fasady ... 109
Wzorzec budowniczego ... 111
Implementacja ... 111
Przemyślenia ... 115
Podsumowanie ... 115

Część II Tworzenie obiektowego zestawu narzędziowego. Proste klasy i interfejsy 117

Rozdział 5. Klasa Collection ... 119
Założenia klasy Collection ... 119
Projektowanie klasy Collection ... 120
Fundamenty klasy Collection ... 121
Metoda addItem ... 122
Metody getItem i removeItem ... 123
Pozostałe metody ... 123
Użycie klasy Collection ... 124
Implementacja leniwej konkretyzacji ... 125
Funkcje zwrotne ... 126
Metoda setLoadCallback w klasie Collection ... 130
Wykorzystanie klasy Collection ... 133
Ulepszanie klasy Collection ... 139
Podsumowanie ... 139

Rozdział 6. Klasa CollectionIterator ... 141
Interfejs Iterator ... 141
Klasa CollectionIterator ... 143
Interfejs IteratorAggregate ... 144
Ochrona zawartości iteratora ... 146
Podsumowanie ... 147

Rozdział 7. Klasa GenericObject ... 149
Klasa GenericObject ... 149
Kiedy korzystać z GenericObject? ... 150
Na co pozwala GenericObject? ... 150
Decyzje w sprawie implementacji ... 151
Typowa implementacja GenericObject ... 152
Poznawanie rodzica ... 154
Współpraca z bazą danych ... 157
Metody i własności GenericObject ... 159
Zalety klasy GenericObject ... 161
Klasa GenericObjectCollection ... 162
Tradycyjna implementacja ... 163
Kiedy tradycyjna implementacja zawodzi ... 163
Działanie klasy GenericObjectCollection ... 164
Kod klasy ... 165
Typowa implementacja GenericObjectCollection ... 167
Próba ... 168
Jak to działa? ... 168
Podsumowanie klasy GenericObjectCollection ... 170
Podsumowanie ... 171

Rozdział 8. Warstwa abstrakcji dla bazy danych ... 173
Czym jest warstwa abstrakcji? ... 174
Prosta implementacja ... 174
Plik konfiguracyjny ... 174
Nawiązywanie połączenia ... 175
Pobieranie danych ... 175
Modyfikacja danych ... 176
Korzystanie z klasy Database ... 178
Wprowadzenie do PEAR DB ... 180
Nawiązywanie połączenia za pomocą DB ... 181
Pobieranie danych ... 182
Inne użyteczne funkcje ... 183
Więcej informacji ... 186
Gotowa warstwa abstrakcji ... 186
Obsługa transakcji ... 189
Wzorzec projektowy Singleton ... 191
Podsumowanie ... 193

Rozdział 9. Interfejs fabryki ... 195
Wzorzec fabryki ... 195
Przykład interfejsu fabryki ... 196
Rozwiązanie staromodne ... 196
Wykorzystanie interfejsu fabryki ... 197
Zastosowanie wzorca w abstrakcji bazy danych ... 198
Większa liczba fabryk ... 200
Wykorzystanie klas istniejących ... 201
Podsumowanie ... 201

Rozdział 10. Programowanie oparte na zdarzeniach ... 203
Czym są zdarzenia? ... 204
Obiektowa obsługa zdarzeń ... 205
Projekt rozwiązania ... 205
Implementacja rozwiązania ... 207
Implementacja zabezpieczeń ... 211
Chwila zastanowienia ... 213
Podsumowanie ... 214

Rozdział 11. Pliki dziennika i debugowanie ... 215
Tworzenie mechanizmu logowania ... 215
Proste logowanie do pliku ... 215
Przykładowa struktura katalogów ... 216
Klasa Logger ... 217
Rozbudowa klasy Logger ... 221
Mechanizm debugowania ... 231
Podsumowanie ... 234

Rozdział 12. SOAP ... 235
SOAP i PHP5 ... 235
Rozszerzenie PHP5 SOAP ... 236
Klient SOAP ... 239
Za kulisami ... 241
Obsługa wyjątków w kliencie SOAP ... 245
Serwer SOAP ... 246
Podsumowanie ... 248

Część III Tworzenie zestawu narzędzi do wielokrotnego wykorzystania. Narzędzia złożone (choć nieskomplikowane) 249

Rozdział 13. Model, widok, kontroler (MVC) ... 251
Wprowadzenie do MVC ... 252
Model ... 253
Widok ... 253
Kontroler ... 253
Infrastruktura ... 253
MVC w aplikacjach WWW ... 253
MVC w PHP ... 254
Mały zestaw narzędzi MVC ... 256
Prezentacja zestawu narzędzi ... 256
Korzystanie z zestawu narzędzi ... 268
Zestaw narzędzi a praktyka ... 275
Prawdziwe szablony ... 275
Powtórka z szablonów macierzystych PHP ... 275
Wady szablonów macierzystych ... 276
Prawdziwe szablony a szablony pakietu Smarty ... 276
Instalowanie pakietu Smarty ... 277
Korzystanie z pakietu Smarty ... 278
Zaawansowane możliwości pakietu Smarty ... 283
Kiedy korzystać z pakietu Smarty, a kiedy z tradycyjnych szablonów? ... 285
Podsumowanie ... 285

Rozdział 14. Komunikacja z użytkownikami ... 287
Po co się komunikować? ... 287
Powody komunikowania się z użytkownikiem ... 288
Myślenie wykraczające poza przeglądarkę WWW ... 290
Formy komunikacji ... 291
Wszystkie formy komunikacji mają… ... 291
Nie wszystkie formy komunikacji mają… ... 291
Co z adresatami? ... 291
Komunikacja jako hierarchia klas ... 292
Klasa adresata — szybki sprawdzian z myślenia obiektowego ... 292
Klasa Communication ... 296
Wysyłanie wiadomości do użytkowników naszej witryny ... 298
Tworzenie wersji testowej ... 298
Wysyłanie wiadomości ... 302
Zastosowanie szablonów przy wykorzystaniu pakietu Smarty ... 307
Korzystanie z MIME ... 309
Inne podklasy klasy Communication ... 309
Wiadomości tekstowe SMS ... 309
Faks ... 310
Podsumowanie ... 310

Rozdział 15. Sesje i uwierzytelnianie ... 311
Wprowadzenie do sesji ... 312
Krótka powtórka z protokołu HTTP ... 312
Definicja sesji ... 314
Ciągłość sesji ... 314
Bezpieczeństwo sesji ... 317
Jak PHP implementuje sesje? ... 324
Podstawowy mechanizm sesji w PHP ... 324
Ograniczenia podstawowego mechanizmu sesji w PHP ... 326
Tworzenie klasy Authentication ... 327
Połączenie zarządzania sesjami PHP z bazą danych ... 327
Klasa UserSession ... 329
Schemat bazy danych ... 329
Kod — usersession.phpm ... 330
Kod — testowanie klasy UserSession ... 334
Jak to działa — klasa UserSession ... 336
Co otrzymaliśmy? ... 339
Podsumowanie ... 339

Rozdział 16. Szkielet do testowania modułów ... 341
Metodologia i terminologia ... 341
Projektowanie interfejsu klasy ... 342
Tworzenie pakietu testowego dla klasy ... 343
Pisanie implementacji naszej klasy ... 344
Druga tura ... 345
Wprowadzenie do PHPUnit ... 345
Instalacja PHPUnit ... 345
Korzystanie z PHPUnit ... 346
Przypadki testowania ... 346
Pakiet testujący ... 349

Czy warto? ... 349
Powtórne testy ... 350
Użyteczność szkieletu ... 350
Demonstrowalny mechanizm zapewniania jakości ... 350
Redukcja obciążenia testami funkcjonalnymi ... 351
Praktyczny przykład ... 351
Podsumowanie ... 356

Rozdział 17. Automat skończony i modyfikowalne pliki konfiguracyjne ... 357

Koncepcja automatu skończonego ... 358
Prosty AS — kalkulator ONP ... 359
Teoretyczna implementacja AS ... 360
Implementacja automatów skończonych w PHP ... 361
Analiza przykładu z kalkulatorem ONP ... 363
Przykłady automatów skończonych w praktyce ... 366
Modyfikowalne pliki konfiguracyjne ... 367
Zastosowanie PHP ... 367
Zastosowanie XML-a ... 368
Korzystanie z plików INI ... 369
Klasa Config z PEAR ... 371
Zalecane praktyki związane z plikami konfiguracyjnymi ... 372
Podsumowanie ... 373

Część IV Studium przypadku — automatyzacja działu sprzedaży 375

Rozdział 18. Przegląd projektu ... 377

Artykulandia ... 378
Krajobraz Artykulandii ... 380
Wymiar techniczny ... 380
Wymiar finansowy ... 380
Wymiar polityczny ... 380
My ... 380
Czy rzeczywiście chodzi o technologię? ... 381
Podejście do budowy oprogramowania ... 381
Jakie konsekwencje ma to dla nas? ... 383
Technologia ... 384
Podsumowanie ... 385

Rozdział 19. Metody zarządzania projektami ... 387

Wstępne rozeznanie ... 387
Dlaczego realizujemy projekt? ... 388
Dla kogo realizujemy projekt? ... 388
Jaka jest historia projektu? ... 390
Jakie są oczekiwane warunki wstępne projektu? ... 390
Odbieranie formalnych wytycznych ... 391
Wymogi obszaru działalności ... 392
Zakres ... 393
Harmonogramy ... 394
Budżet ... 395
Warunki handlowe ... 397
Plany na przyszłość ... 398

Wygląd i obsługa ... 398
Technologia ... 398
Obsługa ... 399
Co dalej? ... 399
Konstruowanie oferty ... 399
Konspekty kontra oferty cenowe ... 399
Oferty w formie konspektu a specyfikacje ... 400
Kogo zaangażować w tworzenie oferty? ... 401
Kiedy można dać z siebie więcej? ... 401
Kiedy powiedzieć „nie”? ... 402
Struktura oferty ... 402
Wybieranie ludzi ... 404
Menedżer projektu ... 404
Account manager ... 404
Główny architekt ... 405
Architekci i inżynierowie oprogramowania ... 406
Programiści interfejsu klienckiego ... 406
Starsi projektanci ... 406
Graficy ... 406
Podwójne role ... 407
Sposób pracy ... 407
Rola klienta ... 407
Podsumowanie ... 408

Rozdział 20. Planowanie systemu ... 409

Wybór procesu ... 409
Proces kaskadowy ... 409
Proces spiralny ... 410
Wybór procesu ... 412
Praktyki wspólne dla obydwu procesów ... 412
Faza specyfikacji ... 412
Faza projektowania ... 415
Faza budowy ... 416
Faza testowania ... 417
Odbiór ... 418
Metodyki i praktyki programowania ... 418
Programowanie inicjowane testami ... 418
Programowanie ekstremalne ... 419
Zarządzanie zmianami ... 422
Rewizje specyfikacji ... 422
Zmiany w specyfikacji, które pojawiają się po jej podpisaniu ... 422
Dyskusje wynikające z różnych interpretacji ... 423
Błędy zgłoszone przez klienta ... 423
Podsumowanie ... 423

Rozdział 21. Architektura systemów ... 425

Czym jest architektura systemu? ... 425
Dlaczego to takie ważne? ... 426
Co musimy zrobić? ... 426
Efektywne tłumaczenie wymagań ... 427
Hosting, łącza, serwery i sieć ... 427
Nadmiarowość i elastyczność ... 428

Utrzymanie ... 428
Bezpieczeństwo ... 429
Projektowanie środowiska ... 429
Hosting i łącza ... 429
Obliczanie parametru CIR ... 430
Serwery ... 432
Sieć ... 434
Zapis nadmiarowy ... 434
Utrzymanie ... 435
Bezpieczeństwo ... 435
Podsumowanie ... 436

Rozdział 22. Tworzenie aplikacji automatyzującej pracę zespołu sprzedaży ... 437

Rozpoczynamy projekt — poniedziałek ... 438
Zamieniamy się w słuch ... 438
Oszacowanie wagi scenariuszy ... 440
Planowanie wersji ... 447
Rozpoczynamy pracę ... 448
Opis szczegółów scenariusza nr 9 ... 448
Tworzenie testów ... 449
PHPUnit ... 450
Tworzenie ekranu logowania ... 457
Następny scenariusz ... 460
Ponowne oszacowanie ... 469
Porządki ... 471
Refaktoring kodu ... 472
Kończenie iteracji ... 478
Scenariusz nr 14. Zmiana tygodnia powoduje odczytanie poprzedniego ... 478
Scenariusz nr 15. Tygodniowe pola na raporcie wizyt klienta ... 480
Raport kosztów podróży ... 487
Składnik kosztów podróży ... 489
Tygodniowe koszty podróży ... 492
Narzut ... 494
Kolejne testy tygodniowych kosztów podróży ... 495
Wypełnianie testów tygodniowego arkusza kosztów podróży ... 498
Zakończony raport kosztów podróży ... 510
Obiekty fikcyjne ... 522
Podsumowanie ... 527

Rozdział 23. Zapewnienie jakości ... 529

Wprowadzenie do QA ... 529
Dlaczego warto się starać? ... 530
Co to jest jakość? ... 531
Wymierna jakość ... 532
Testowanie ... 534
Testowanie modułów ... 535
Testowanie funkcjonalne ... 535
Testowanie obciążenia ... 537
Testowanie użyteczności ... 537
Śledzenie błędów ... 538
Efektywne śledzenie błędów z wykorzystaniem systemu Mantis ... 539
Wykorzystanie wszystkich możliwości Mantis ... 546
Podsumowanie ... 546

Rozdział 24. Instalacja ... 549
Opracowywanie środowiska programistycznego ... 549
Firmowe środowisko rozwojowe ... 550
Firmowe środowisko testowe ... 551
Środowisko testowe klienta ... 551
Środowisko produkcyjne klienta ... 552
Rozwojowe bazy danych ... 553
Organizacja wdrożenia ... 554
Automatyczne pobieranie z repozytorium kontroli wersji ... 556
Zastosowanie rsync ... 557
Synchronizacja serwerów za pomocą rsync ... 559
Podsumowanie ... 561

Rozdział 25. Projektowanie i tworzenie solidnej platformy raportującej ... 563
Wprowadzenie do danych roboczych ... 563
Poznajemy potrzeby klienta ... 564
Zarządzanie żądaniami klientów ... 564
Dostarczanie raportów ... 566
Projektowanie raportu ... 566
Architektura generatora raportów ... 569
Generowanie raportów w tle ... 571
Interfejs raportów ... 573
Interfejs nowego raportu ... 574
Skrypt procesora raportów ... 578
Proces ... 578
Skrypty obsługi raportów ... 579
Strona Moje raporty ... 581
Skrypty tłumaczące ... 581
Przykład użycia platformy raportowej ... 583
Wizualizacja ... 584
Podsumowanie ... 585

Rozdział 26. Co dalej? ... 587
Motywacja ... 587
Twoja kariera programisty ... 588
Więcej niż tylko programowanie WWW ... 588
Umiejętności miękkie ... 589
Umiejętności teoretyczne ... 589
Umiejętności społeczne ... 589
Podsumowanie ... 590

Dodatki 591

Dodatek A Dlaczego warto korzystać z kontroli wersji ... 593

Dodatek B IDE dla PHP ... 607

Dodatek C Strojenie wydajności PHP ... 621

Dodatek D Najlepsze praktyki przy instalacji PHP ... 633

Skorowidz ... 645

1

Wprowadzenie do programowania obiektowego

Programowanie obiektowe może wprowadzać zamieszanie w głowach programistów tworzących głównie kod proceduralny. Może, ale nie musi. W niniejszym rozdziale omówimy podstawowe zagadnienia teoretyczne związane z technologią obiektową i poznamy obowiązującą w tej dziedzinie terminologię (pełną odstraszających czasem wielosylabowców). Powiemy, dlaczego warto interesować się technikami obiektowymi i w jaki sposób mogą one znacznie przyspieszyć proces programowania rozbudowanych aplikacji oraz ułatwić ich późniejsze modyfikacje.

W dwóch następnych rozdziałach będziemy poszerzać tę wiedzę i wgłębiać się w nieco bardziej zaawansowane tematy. Ci, którzy mają wcześniejsze doświadczenia z programowaniem obiektowym poza środowiskiem PHP5, mogą te dwa rozdziały pominąć. Z drugiej strony, materiał ten może stanowić dobrą powtórkę, więc mimo wszystko zachęcamy do jego przeczytania.

Czym jest programowanie obiektowe?

Programowanie obiektowe wymaga innego sposobu myślenia przy tworzeniu aplikacji. Obiekty umożliwiają bliższe odwzorowanie za pomocą kodu, rzeczywistych zadań, procesów i idei, które mają realizować aplikacje. Zamiast traktować aplikację jako wątek sterowania, który przesyła dane pomiędzy funkcjami, modelujemy ją jako zbiór współpracujących z sobą obiektów, które niezależnie realizują pewne zadania.

Analogią może być przykład budowy domu. Hydraulicy zajmują się instalacją wodną, a elektrycy instalacją elektryczną. Hydraulicy nie muszą wiedzieć, czy obwód w sypialni jest 10-amperowy czy 20-amperowy. Interesuje ich tylko to, co ma jakiś związek z instalacją wodną. Generalny wykonawca dopilnowuje tego, by każdy z podwykonawców zrobił to, co do niego należy, ale przeważnie nie interesują go szczegóły każdego pojedynczego zadania.

Podejście obiektowe jest podobne w tym sensie, że obiekty ukrywają przed sobą szczegóły swojej implementacji. Sposób wykonania zadania nie jest istotny dla innych komponentów systemu. Liczy się to, jakie usługi obiekt może wykonać.

Pojęcia klas i obiektów oraz sposoby korzystania z nich przy pisaniu programów to fundamentalne zagadnienia programowania obiektowego. Są one w pewnym sensie sprzeczne z zasadami programowania proceduralnego, czyli programowania korzystającego z funkcji i globalnych struktur danych. Jak zobaczymy, podejście obiektowe ma kilka ogromnych zalet w porównaniu z proceduralnym, a nowa implementacja możliwości programowania obiektowego w PHP5 przynosi również znaczne korzyści w wydajności.

Zalety programowania obiektowego

Jedną z głównych zalet programowania obiektowego jest łatwość przekładania poszczególnych wymogów z obszaru zastosowania na poszczególne moduły kodu. Ponieważ podejście obiektowe pozwala na modelowanie aplikacji na podstawie „obiektów" z otaczającej nas rzeczywistości, często możliwe jest odnalezienie bezpośredniego przełożenia ludzi, rzeczy i pojęć na odpowiadające im klasy. Klasy te charakteryzują się takimi samymi właściwościami i zachowaniami jak rzeczywiste pojęcia, które reprezentują, co pomaga w szybkim ustaleniu, jaki kod trzeba napisać i w jaki sposób zachodzić ma interakcja pomiędzy poszczególnymi częściami aplikacji.

Drugą zaletą programowania obiektowego jest możliwość wielokrotnego wykorzystania kodu. Często potrzebujemy tych samych typów danych w różnych miejscach tej samej aplikacji. Na przykład w aplikacji, która umożliwia szpitalowi zarządzanie kartami swoich pacjentów, z pewnością potrzebna będzie klasa `Person` (osoba). W systemie opieki szpitalnej występuje wiele różnych osób — pacjent, lekarze, pielęgniarki, członkowie administracji szpitala itp. Na każdym etapie zajmowania się pacjentem, wymagane jest odnotowanie w jego karcie osoby, która wykonała daną czynność (np. przepisała lek, oczyściła ranę lub wystawiła rachunek za opiekę medyczną), i zweryfikowanie, czy osoba jest do tej czynności uprawniona. Definiując ogólną klasę `Person`, która obejmuje wszystkie właściwości i metody wspólne dla wszystkich ludzi, otrzymujemy możliwość wykorzystania tego fragmentu kodu na naprawdę olbrzymią skalę, co nie zawsze jest możliwe przy proceduralnym podejściu do programowania.

A co z innymi zastosowaniami? Czy możemy wyobrazić sobie inne aplikacje, które przetwarzają informacje o osobach? Prawdopodobnie tak i to niemało. Dobrze napisana klasa `Person` mogłaby być przenoszona z jednego projektu do drugiego z minimalnymi zmianami lub wręcz bez zmian, udostępniając od ręki duże możliwości funkcjonalne stworzone przy okazji wcześniejszych prac. Możliwość wielokrotnego wykorzystania kodu zarówno w obrębie jednej aplikacji, jak i od projektu do projektu, to jedna z większych zalet podejścia obiektowego.

Inna zaleta programowania obiektowego wynika z modularności klas. Jeżeli odkryjemy błąd w klasie `Person` albo będziemy chcieli zmienić sposób jej działania czy też rozbudować ją, to wszelkich zmian dokonujemy tylko w jednym miejscu. Wszystkie cechy funkcjonalne klasy znajdują się w jednym pliku. Dokonane zmiany od razu odzwierciedlają się we wszystkich procesach aplikacji, które bezpośrednio opierają się na klasie `Person`. Znacznie upraszcza to szukanie błędów i czyni rozbudowę klasy zabiegiem w miarę bezbolesnym.

Przykład z życia

Korzyści wynikające z modularności mogą wydawać się niewielkie w przypadku prostych aplikacji, ale podczas wykorzystywania złożonych architektur oprogramowania bywają potężne. Jeden z autorów książki pracował ostatnio nad projektem obejmujących 200 tysięcy wierszy proceduralnego kodu PHP. Co najmniej 65% czasu poświęconego na poprawianie błędów zmarnowano na wyszukiwanie pewnych funkcji i sprawdzanie, jakie funkcje korzystają z jakich danych. Później stworzono nową wersję tego oprogramowania, tym razem o architekturze obiektowej, która okazała się składać z o wiele mniejszej ilości kodu. Gdyby aplikacja od początku była pisana w taki sposób, to nie tylko trwałoby to krócej, ale uniknięto by wielu błędów (im mniejsza ilość kodu, tym mniejsza ilość błędów), a proces ich usuwania byłby znacznie szybszy.

Jako że podejście obiektowe zmusza do zastanowienia się nad organizacją kodu, poznawanie struktury istniejącej aplikacji jest o wiele łatwiejsze, kiedy dołączamy jako „nowy" do zespołu programistów. Poza tym dysponujemy wówczas szkieletem pomocnym w ustalaniu, gdzie powinny znaleźć się nowo dodawane cechy funkcjonalne.

Nad większymi projektami często pracują wieloosobowe zespoły programistów o różnych umiejętnościach. Tutaj również podejście obiektowe ma znaczną przewagę nad proceduralnym. Obiekty ukrywają szczegóły implementacyjne przed swoimi użytkownikami. Zamiast konieczności zrozumienia złożonych struktur danych i pokrętnej logiki rządzącej obszarem zastosowania aplikacji mniej doświadczeni członkowie zespołu mogą na podstawie skromnej dokumentacji zacząć używać obiektów stworzonych przez bardziej doświadczonych programistów. Same obiekty są odpowiedzialne za dokonywanie zmian w danych oraz zmian stanu systemu.

Kiedy wcześniej wspomniana aplikacja wciąż jeszcze była pisana kodem proceduralnym, nowi programiści w zespole często musieli poświęcić do dwóch miesięcy na naukę szczegółów związanych z aplikacją, zanim stali się produktywni. Po przerobieniu aplikacji na obiektową nowi członkowie zespołu mogli już po kilku dniach tworzyć obszerne dodatki do istniejącej bazy kodu. Byli w stanie szybko nauczyć się korzystać nawet z najbardziej skomplikowanych obiektów, ponieważ nie musieli w pełni rozumieć wszystkich szczegółów implementacji reprezentowanych przez nie cech funkcjonalnych.

Teraz, gdy już wiemy, dlaczego należy rozważyć zastosowanie paradygmatu obiektowego jako metody programowania, powinniśmy przeczytać kilka następnych podrozdziałów, aby lepiej zrozumieć koncepcje leżące u podstaw obiektowości. Potem, już w trakcie lektury dwóch kolejnych rozdziałów, powinniśmy na własnej skórze odczuć zalety tego podejścia.

Pojęcia związane z programowaniem obiektowym

W niniejszym podrozdziale wprowadzimy podstawowe pojęcia ze świata programowania obiektowego i poznamy zależności miedzy nimi. W rozdziale 3., „Obiekty zaczynają działać", omówiono specyfikę implementacji tych pojęć w PHP5. Poznamy tu między innymi:

- **Klasy** — „wzorce” dla obiektów i kod definiujący właściwości i metody.
- **Obiekty** — stworzone egzemplarze klasy, które przechowują wszelkie wewnętrzne dane i informacje o stanie potrzebne dla funkcjonowania aplikacji.
- **Dziedziczenie** — możliwość definiowania klas jednego rodzaju jako szczególnego przypadku (podtypu) klasy innego rodzaju (na podobnej zasadzie jak kwadrat określany jest jako szczególny przypadek prostokąta).
- **Polimorfizm** — umożliwia zdefiniowanie klasy jako członka więcej niż jednej kategorii klas (tak jak samochód, który jest „czymś, co ma silnik” oraz „czymś, co ma koła”).
- **Interfejsy** — stanowią „umowę” na podstawie której obiekt może implementować metodę, nie definiując rzeczywistego sposobu implementacji.
- **Hermetyzacja** — możliwość zastrzeżenia dostępu do wewnętrznych danych obiektu.

Nie należy się przejmować, jeżeli któryś z tych terminów wydaje się trudny do zrozumienia. Wszystkie zostaną wyjaśnione dalej. Nowo zdobyta wiedza może całkowicie zmienić nasze podejście do realizacji przedsięwzięć programistycznych.

Klasy

W otaczającej nas rzeczywistości obiekty mają pewną charakterystykę i zachowania. Samochód ma kolor, wagę, markę oraz bak paliwowy o pewnej pojemności. To jest jego charakterystyka. Samochód może przyspieszać, zatrzymać się, sygnalizować skręt lub trąbić klaksonem. To są jego zachowania. Te cechy i zachowania są wspólne dla wszystkich samochodów. Co prawda różne samochody mają różne kolory, ale każdy samochód ma jakiś kolor. W programowaniu obiektowym **klasa** umożliwia ustanowienie pojęcia samochodu jako czegoś posiadającego wszystkie cechy uznane za wspólne. Klasa to zamknięty fragment kodu złożony ze zmiennych i funkcji, które opisują cechy oraz zachowania wspólne dla wszystkich elementów pewnego zbioru. Klasa o nazwie `Car` (samochód) opisywałaby właściwości i metody wspólne dla wszystkich samochodów.

W terminologii obiektowej charakterystyki klasy określa się mianem **właściwości**. Właściwości mają nazwę i wartość. Wartości niektórych można zmieniać, a innych nie. Na przykład w klasie `Car` wystąpiłyby zapewne takie właściwości jak `color` (kolor) czy `weigth` (waga). Kolor samochodu może ulec zmianie po lakierowaniu, ale waga samochodu (bez pasażerów i bagażu) jest wartością stałą.

Niektóre właściwości reprezentują stan obiektu. Stan odnosi się do tych charakterystyk, które ulegają zmianie w efekcie pewnych zdarzeń, a niekoniecznie można je modyfikować bezpośrednio. W aplikacji, która symuluje funkcjonowanie samochodu, klasa `Car` może mieć właściwość `velocity` (prędkość). Prędkość nie jest wartością, którą można zmienić tak po prostu, lecz końcowym efektem ilości paliwa przesłanej do silnika, osiągów tego silnika oraz terenu, po jakim porusza się samochód.

Zachowania klas są określane mianem **metod**. Metody klas są składniowymi odpowiednikami funkcji z tradycyjnych programów proceduralnych. Podobnie jak funkcje, metody mogą pobierać dowolną ilość parametrów, z których każdy jest pewnego dopuszczalnego typu.

Niektóre metody przetwarzają zewnętrzne dane, przesłane jako parametry, ale mogą również działać na właściwościach własnych obiektów, odczytując ich wartości na potrzeby wykonywanych działań (na przykład metoda `accelerate`, która symuluje naciśnięcie pedału gazu, może sprawdzać ilość pozostałego paliwa, by ustalić, czy przyspieszenie jest możliwe) albo zmieniając stan obiektów poprzez modyfikację takich wartości jak prędkość samochodu.

Obiekty

Na początek klasę można potraktować jako wzorzec, na postawie którego konstruowany jest obiekt. Podobnie jak na podstawie tego samego projektu (wzorca) można zbudować wiele domów, tak samo możliwe jest stworzenie wielu egzemplarzy obiektów jednej klasy. Jednak projekt domu nie precyzuje takich szczegółów jak kolor ścian czy rodzaj posadzki, ustalając jedynie, że takie rzeczy istnieją. Klasy funkcjonują podobnie, określając zachowania i charakterystyki obiektu, nie przesądzając o ich konkretnej wartości lub stanie. Obiekt to element konkretny skonstruowany na podstawie wzorca dostarczonego przez klasę. Ogólne pojęcie „dom" można porównać do klasy. Z kolei „nasz dom" (określony reprezentant pojęcia „dom") można porównać do obiektu.

Mając projekt czy wzorzec i jakieś materiały budowlane, możemy zbudować dom. W programowaniu obiektowym, aby zbudować obiekt, posługujemy się klasą. Proces ten nazywa się **tworzeniem egzemplarza** i wymaga dwóch rzeczy:

- Miejsca w pamięci przeznaczonego dla obiektu. Tym akurat PHP zajmie się automatycznie.
- Danych, które zostaną przypisane wartościom właściwości. Dane te mogą pochodzić z bazy danych, pliku tekstowego, innego obiektu lub jeszcze innego źródła.

Sama klasa nie może mieć przypisanych wartości do właściwości albo być w jakimś stanie. Mogą to jedynie obiekty. Aby zbudować dom, trzeba posłużyć się projektem. Dopiero potem możemy pomyśleć o tapetowaniu i panelach zewnętrznych. Podobnie konieczne jest stworzenie egzemplarza obiektu klasy, zanim będziemy mogli operować na jego właściwościach lub wywoływać jego metody. Klasami manipulujemy w czasie pisania programu, modyfikując kod metod i właściwości. Obiektami manipulujemy w trakcie wykonywania programu, przypisując wartości właściwościom i wywołując metody. Początkujący adepci programowania obiektowego często nie są pewni, kiedy powinni posługiwać się pojęciem klasy, a kiedy obiektu.

Po utworzeniu obiektu można go przystosować tak, by implementował wymogi obszaru zastosowania aplikacji. Przyjrzyjmy się dokładnie, jak to się robi w PHP.

Tworzenie klasy

Zacznijmy od prostego przykładu. Zapisz poniższy kod w pliku o nazwie *class.Demo.php*:

```
<?php
   class Demo {
   }
?>
```

I już. Właśnie stworzyliśmy klasę `Demo`. Co prawda nie wygląda imponująco, ale to nic innego jak podstawowa składnia deklarowania nowej klasy w PHP. Używamy słowa kluczowego `class`, aby poinformować PHP, że mamy zamiar zdefiniować nową klasę. Po nim podajemy nazwę klasy, a treść klasy zawieramy pomiędzy klamrą otwierającą a zamykającą.

> Ważne jest zdefiniowanie jasnej konwencji organizowania plików z kodem źródłowym. Dobrą zasadą jest umieszczanie każdej klasy w osobnym pliku i nadanie mu nazwy *class.[NazwaKlasy].php*.

Obiekt typu `Demo` można utworzyć tak:

```
<?php
   require_once('class.Demo.php');
   $objDemo = new Demo();
?>
```

Aby utworzyć obiekt, najpierw należy się upewnić, czy PHP wie, gdzie odnaleźć deklarację klasy poprzez dołączenie pliku z treścią klasy (w tym przykładzie jest to *class.Demo.php*), potem wywołać operator `new` i podać nazwę klasy oraz parę nawiasów. Wartość zwracana przez tę instrukcję zostaje przypisana do nowej zmiennej `$objDemo`. Teraz można już wywoływać metody obiektu `$objDemo` i odczytywać lub ustawiać jego właściwości — o ile został w takowe wyposażony.

Mimo że klasa, którą właśnie stworzyliśmy, w zasadzie nic nie robi, wciąż stanowi poprawną definicję klasy.

Dodawanie metody

Klasa `Demo` nie będzie zbyt przydatna, jeżeli nie będzie nic robić. Przyjrzyjmy się więc, jak stworzyć metodę. Pamiętajmy, że metoda klasy to po prostu funkcja. Pisząc treść funkcji pomiędzy klamrami otwierającymi i zamykającymi definicję klasy, dodajemy do klasy nową metodę. Oto przykład:

```
<?php
   class Demo {
      function sayHello($name) {
         print "Cześć, $name!";
      }
   }
?>
```

Obiekt stworzony na postawie klasy może teraz wyświetlić pozdrowienia dla każdego, kto wywoła metodę `sayHello`. Aby wywołać tę metodę na obiekcie `$objDemo`, konieczne jest zastosowanie operatora `->` umożliwiającego dostęp do utworzonej funkcji:

```
<?php
   require_once('class.Demo.php');
   $objDemo = new Demo();
   $objDemo->sayHello('Stefan');
?>
```

Obiekt jest teraz w stanie wyświetlić tekst przyjaznego pozdrowienia. Operator `->` służy do dostępu do wszystkich metod i właściwości obiektów.

Tym, którzy mieli do czynienia z programowaniem obiektowym w innych językach, zwracamy uwagę, że dostęp do metod i właściwości obiektu jest realizowany za pomocą operatora `->`. Operator kropki (`.`) nie występuje w składni PHP w ogóle.

Dodawanie właściwości

Dodawanie właściwości do klasy jest równie proste jak dodawanie metody. Wystarczy zadeklarować zmienną w obrębie klasy, która będzie przechowywać wartość właściwości. W kodzie proceduralnym, jeżeli chcieliśmy przechować jakąś wartość, to przypisywaliśmy ją zmiennej. W programowaniu obiektowym do przechowywania wartości właściwości również używamy zmiennej. Zmienna ta jest deklarowana na początku deklaracji klasy, w obrębie klamer, które zamykają w sobie kod klasy. Nazwa zmiennej jest nazwą właściwości. Jeżeli zmienna nazywa się `$color`, to tworzymy właściwość zwaną `color`.

Otwórzmy plik *class.Demo.php* i zastąpmy jego treść następującym kodem:

```
<?php
   class Demo {
      public $name;
      function sayHello() {
         print "Cześć, $this->name!";
      }
   }
?>
```

Deklaracja nowej zmiennej `$name` to wszystko, czego trzeba, by stworzyć właściwość klasy `Demo` zwaną `name`. Dostęp do tej właściwości wymaga posłużenia się tym samym operatorem (`->`) co w poprzednim przykładzie oraz nazwą właściwości. Nowa wersja metody `sayHello` ukazuje, jak realizujemy dostęp do tej właściwości.

Stwórzmy nowy plik o nazwie *testdemo.php* i wpiszmy w nim następujący kod:

```
<?php

   require_once('class.Demo.php');

   $objDemo = new Demo();
   $objDemo->name = 'Stefan';

   $objAnotherDemo = new Demo();
   $objAnotherDemo->name = 'Edek';

   $objDemo->sayHello();
   $objAnotherDemo->sayHello();

?>
```

Po zapisaniu pliku otwórzmy go w przeglądarce WWW. Na ekranie wyświetlą się ciągi „Cześć, Stefan!” i „Cześć, Edek!”.

Słowo kluczowe `public` służy do określania, że chcemy mieć dostęp spoza klasy do następującej po nim zmiennej. Niektóre zmienne składające się na klasę istnieją tylko na potrzeby samej klasy i nie powinny być dostępne z poziomu zewnętrznego kodu. W tym przykładzie

chcemy mieć możliwość ustawiania i pobierania wartości właściwości `name`. Jak widać, sposób działania metody `sayHello` nieco się zmienił. Zamiast pobierać parametr, pobiera ona teraz wartość `name` wprost z właściwości.

Posługujemy się zmienną `$this`, aby obiekt pobierał informację dotyczącą jego samego. Czasami istnieje kilka obiektów danej klasy i, jako że nie znamy z góry nazwy zmiennej reprezentującej obiekt, zmienna `$this` pozwala na odwołanie się do bieżącego egzemplarza.

W poprzednim przykładzie pierwsze wywołanie `sayHello` wyświetla imię `Stefan`, a drugie — `Edek`. To dlatego, że zmienna `$this` umożliwia każdemu obiektowi dostęp do własnych właściwości i metod bez znajomości nazwy, która reprezentuje ten obiekt na zewnątrz klasy. Wcześniej wspomnieliśmy, że niektóre właściwości mają wpływ na działanie pewnych metod, powołując się na przykład metody `accelerate` w klasie `Car`, która sprawdza ilość pozostałego paliwa. W treści metody `accelerate` dostęp do odpowiedniej właściwości zrealizowano by zapewne wywołaniem `$this->amountOfFuel`.

> Dostęp do właściwości wymaga tylko jednego `$`. Składnia to `$obiekt->wlasciwosc`, a nie `$obiekt->$wlasciwosc`. Często myli to stawiających pierwsze kroki w PHP. Zmienna reprezentująca właściwość jest deklarowana jako `public $wlasciwosc`, a dostęp wymaga wywołania postaci `$obiekt->wlasciwosc`.

Poza zmiennymi przechowującymi wartości właściwości klasy można deklarować inne zmienne przeznaczone do wewnętrznego użytku w klasie. Obydwa rodzaje danych są wspólnie nazywane **wewnętrznymi zmiennymi składowymi** klasy. Część z nich jest dostępna spoza klasy jako właściwości, a inne nie są dostępne i służą tylko wewnętrznym potrzebom klas. Jeżeli, na przykład, klasa `Car` musiałaby pobrać z jakiegoś powodu informację z bazy danych, mogłaby zachowywać uchwyt do połączenia z bazą danych w wewnętrznej zmiennej składowej. Sam uchwyt do połączenia z bazą danych trudno nazwać właściwością samochodu — jest jedynie czymś, co umożliwia klasie wykonanie pewnych operacji.

Ochrona dostępu do zmiennych składowych

Jak ukazuje poprzedni przykład, właściwości `name` można przypisać, co tylko zechcemy, w tym obiekt, tablicę liczb całkowitych, uchwyt pliku lub każdą inną bezsensowną wartość. Nie mamy jednak możliwości przeprowadzenia jakiejkolwiek kontroli poprawności danych ani zaktualizowania jakichkolwiek innych wartości w chwili ustawiania wartości właściwości `name`.

Aby to obejść, należy zawsze implementować właściwości jako funkcje wywoływane jako `get[NazwaWlasciwosci]` i `set[NazwaWlasciwosci]`. Funkcje te znane są jako **metody dostępowe** — zostaną zaprezentowane w poniższym przykładzie.

Wprowadźmy w pliku *class.Demo.php* następujące zmiany:

```
<?php

   class Demo {

      private $_name;

      public function sayHello() {
```

```
            print "Cześć, {$this->getName()}!";
        }

        public function getName() {
            return $this->_name;
        }

        public function setName($name) {
            if(!is_string($name) || strlen($name) == 0) {
                throw new Exception("Niepoprawna wartość zmiennej name");
            }

            $this->_name = $name;
        }

    }
?>
```

Plik *testdemo.Php* zmodyfikujmy następująco:

```
<?php

    require_once('class.Demo.php');

    $objDemo = new Demo();
    $objDemo->setName('Stefan');
    $objDemo->sayHello();

    $objDemo->setName(37); //wygeneruje błąd

?>
```

Jak widać, status dostępu do składnika zmienił się z `public` na `private`, a jego nazwa została poprzedzona podkreśleniem. Podkreślenie jest zalecaną konwencją nazewniczą zmiennych prywatnych i funkcji składowych, ale tylko konwencją i PHP nie wymaga jej stosowania. Słowo kluczowe `private` uniemożliwia modyfikację tej wartości spoza poziomu tego obiektu. Prywatne, wewnętrzne zmienne składowe nie są dostępne spoza klasy. Ponieważ nie ma bezpośredniego dostępu do zmiennych, trzeba zdać się na pośrednictwo metod dostępowych `getName()` i `setName()`, dzięki czemu klasa będzie mogła sprawdzić prawidłowość wartości, zanim zostanie ona przypisana. W tym przykładzie, w przypadku przesłania nieprawidłowej wartości dla `name`, generowany jest wyjątek. Dodano też specyfikator `public` dla funkcji. Poziom publiczny to domyślny poziom dostępności dla każdej funkcji lub zmiennej składowej, które jawnie go nie ustawiają, ale dobrym nawykiem jest jawne podawanie statusu dostępności dla wszystkich składowych klasy.

Zmienna składowa lub metoda może mieć trzy różne poziomy dostępności: publiczny, prywatny i chroniony. Składowe publiczne są dostępne z poziomu dowolnego kodu, a składowe prywatne dostępne są tylko z poziomu klasy. Te ostatnie to zwykle elementy o zastosowaniu wewnętrznym, takie jak uchwyt połączenia z bazą danych czy informacje konfiguracyjne. Składowe chronione są dostępne dla samej klasy oraz dla klas jej potomnych (dziedziczenie zostało zdefiniowane i opisane w dalszej części rozdziału).

Dzięki stworzeniu metod dostępowych dla wszystkich właściwości o wiele łatwiej dodać testowanie prawidłowości danych, nową logikę obszaru zastosowania lub inne przyszłe zmiany w obiektach. Nawet jeżeli bieżące wymogi wobec aplikacji nie narzucają konieczności

sprawdzania prawidłowości danych dla pewnej właściwości, należy mimo to zaimplementować ją, posługując się funkcjami `get` i `set`, aby możliwe było dodanie takiego sprawdzania, albo zmianę logiki obszaru zastosowania w przyszłości.

> Zawsze używajmy metod dostępowych dla właściwości. Ułatwi to implementację przyszłych zmian w wymaganiach logiki obszaru zastosowania aplikacji i kontroli prawidłowości zmiennych.

Inicjalizacja obiektów

Dla wielu stworzonych klas potrzebna będzie specjalna procedura inicjalizacyjna wykonywana w chwili tworzenia egzemplarza obiektu tej klasy. Konieczne może być, na przykład, pobranie jakichś danych z bazy lub zainicjalizowanie jakichś właściwości. Tworząc specjalną metodę, zwaną **konstruktorem**, zaimplementowaną w PHP funkcją `__construct()`, możemy wykonać wszelkie czynności konieczne dla zainicjalizowania obiektu. PHP automatycznie wywoła tę funkcję w chwili tworzenia egzemplarza obiektu.

Możemy, na przykład, napisać następującą, nową wersję klasy `Demo`:

```
<?php

   class Demo {

      private $_name;

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

      function sayHello() {
         print "Cześć, $this->name!";
      }
   }
?>
```

Funkcja `__construct` zostanie automatycznie wywołana w chwili tworzenia nowego egzemplarza klasy `Demo`.

> Uwaga dla użytkowników PHP4: w PHP4 konstruktory obiektów były funkcjami o nazwach takich samych jak nazwa klasy. W PHP5 zdecydowano się na zastosowanie jednorodnej formy konstruktora. Dla zachowania kompatybilności wstecz PHP najpierw szuka funkcji o nazwie `__construct`, po czym, jeżeli takiej nie znajdzie, szuka jeszcze funkcji o nazwie takiej samej jak nazwa klasy (w poprzednim przykładzie byłaby to `public function Demo()`).

Jeżeli mamy klasę, która nie wymaga żadnej szczególnej procedury inicjalizacyjnej, aby móc normalnie działać, nie trzeba tworzyć konstruktora. Jak widzieliśmy w pierwszej wersji klasy `Demo`, PHP automatycznie robi wszystko, co potrzeba, by utworzyć sam obiekt. Konstruktory należy pisać tylko wtedy, gdy są potrzebne.

Likwidowanie obiektów

Zmienne obiektów, które tworzymy, są usuwane z pamięci systemowej w chwili zakończenia wykonywania kodu bieżącej strony, kiedy zmienna zniknie z bieżącego zakresu lub kiedy jawnie zostanie jej przypisana wartość pusta. W PHP5 można uchwycić moment likwidacji obiektu, by wykonać w tym czasie pewne czynności. W tym celu należy stworzyć funkcję zwaną `__destruct` nie pobierającą parametrów. Funkcja ta, o ile istnieje, jest wywoływana, zanim obiekt zostanie zlikwidowany.

Poniższy przykład zapisuje w chwili likwidacji obiektu informację w pliku dziennika zdarzeń o tym, jak długo obiekt istniał. Jeżeli mamy obiekty, które szczególnie obciążają pamięć lub procesor, to taka technika może przydać się w analizie wydajności systemu i przy szukaniu sposobów redukcji nadmiernego obciążenia.

Tak jak w większości podanych w tej książce przykładów korzystających z baz danych platformą jest tu PostgreSQL. Zdaniem autorów zaawansowane funkcje, obsługa transakcji i sprawny mechanizm procedur składowanych tej bazy czynią z niej lepszą alternatywę dla MySQL-a i innych relacyjnych baz danych typu *open-source* przeznaczonych do tworzenia dużych, profesjonalnych baz danych. Czytelnicy, którzy nie dysponują środowiskiem PostgeSQL, mogą dokonać odpowiednich modyfikacji dostosowujących do używanej platformy bazodanowej.

Stwórzmy tabelę, zwaną `artykul`, posługując się następującą instrukcją SQL:

```
CREATE TABLE "artykul" (
   "nr_artykulu" SERIAL PRIMARY KEY NOT NULL,
   "nazwa" varchar(255) NOT NULL,
   "opis" text
);
```

Wprowadźmy do niej jakieś dane:

```
INSERT INTO "artykul" ("nazwa", "opis")
VALUES('Korale', 'Korale koloru koralowego');
```

Utwórzmy plik o nazwie *class.Widget.php* i wprowadźmy w nim następujący kod:

```
<?php

class Widget {

   private $id;
   private $name;
   private $description;  private $hDB;
   private $needsUpdating = false;

   public function __construct($widgetID) {
      //Parametr widgetID to klucz główny rekordu
      //w bazie danych zawierającego dane
      //tego obiektu

      //Tworzy uchwyt połączenia i zachowuje go w prywatnej zmiennej składowej
      $this->hDB = pg_connect('dbname=parts user=postgres');
      if(! is_resource($this->hDB)) {
```

```
            throw new Exception('Nie można połączyć się z bazą danych.');
         }

         $sql = "SELECT \"nazwa\", \"opis\" FROM artykul WHERE nr_artykulu = $widgetID";
         $rs = pg_query($this->hDB, $sql);
         if(! is_resource($rs)) {
            throw new Exception("Błąd przy wykonywaniu instrukcji wyboru.");
         }

         if(! pg_num_rows($rs)) {
            throw new Exception('Szukanego artykułu nie ma w bazie!');
         }

         $data = pg_fetch_array($rs);
         $this->id = $widgetID;
         $this->name = $data['nazwa'];
         $this->description = $data['opis'];

      }

      public function getName() {
         return $this->name;
      }

      public function getDescription() {
         return $this->description;
      }

      public function setName($name) {
         $this->name = $name;
         $this->needsUpdating = true;
      }

      public function setDescription($description) {
         $this->description = $description;
         $this->needsUpdating = true;
      }

      public function __destruct() {
         if(! $this->needsUpdating) {
            return;
         }

         $sql = 'UPDATE "artykul" SET ';
         $sql .= "\"nazwa\" = '" . pg_escape_string($this->name) . "', ";
         $sql .= "\"opis\" = '" . pg_escape_string($this->description) . "' ";
         $sql .= "WHERE nr_artykulu = " . $this->id;

         $rs = pg_query($this->hDB, $sql);
         if(! is_resource($rs)) {
            throw new Exception('Wystąpił błąd podczas aktualizacji bazy danych');
         }

         //Koniec operacji na bazie danych. Zamknięcie połączenia.
         pg_close($this->hDB);
      }
   }
   ?>
```

Konstruktor tego obiektu nawiązuje połączenie z bazą danych `parts`, posługując się kontem domyślnego superużytkownika `postgres`. Uchwyt do tego połączenia zostaje zachowany jako prywatna zmienna składowa do późniejszego użycia. Wartość `$widgetID` przesłana jako parametr do konstruktora służy do skonstruowania instrukcji SQL, która pobiera informacje o artykule przechowywanym w bazie pod podaną wartością klucza głównego. Dane z bazy są wówczas przypisywane do prywatnych zmiennych składowych, z których można korzystać za pomocą funkcji `get` i `set`. Zwróćmy uwagę, że w razie jakiegokolwiek niepowodzenia konstruktor generuje wyjątek, należy więc pamiętać o zawieraniu wszelkich prób konstruowania obiektów `Widget` w blokach `try...catch`.

Dwie metody dostępowe, `getName()` i `getDescription()`, umożliwiają pobieranie wartości prywatnych zmiennych składowych. Podobnie `setName()` i `setDescription()` umożliwiają przypisywanie im nowych wartości. Zauważmy, że w chwili przypisywania nowej wartości zmienna `needsUpdating` jest ustawiana na `true`. Jeżeli nie nastąpiły żadne zmiany, nic nie musi być aktualizowane.

Dla testu stworzymy plik o nazwie *testWidget.php* i wpiszemy do niego następującą treść:

```
<?php

require_once('class.Widget.php');

   try {
      $objWidget = new Widget(1);

      print "Nazwa artykułu: " . $objWidget->getName() . "<br>\n";
      print "Opis artykułu: " . $objWidget->getDescription() . "<br>\n";

      $objWidget->setName('Trampki');
      $objWidget->setDescription('Trampki o dużym przebiegu!');
   } catch (Exception $e) {
      die("Wystąpił problem: " . $e->getMessage());
   }

?>
```

Spróbujmy uruchomić ten plik w przeglądarce WWW. Po pierwszym wywołaniu powinien wygenerować następującą treść:

```
Nazwa artykułu: Korale
Opis artykułu: Korale koloru koralowego
```

Każde kolejne wywołanie powinno wyświetlić:

```
Nazwa artykułu: Trampki
Opis artykułu: Trampki o dużym przebiegu!
```

Zobaczmy, jak ogromne możliwości ma ta technika. Pobieramy obiekt z bazy danych, zmieniamy właściwość tego obiektu i „automagicznie” zapisujemy zmienione dane w bazie za pomocą kilku zaledwie wierszy kodu w *testWidget.php*. Jeżeli nic się nie zmienia, powrót do bazy danych nie jest potrzebny i dzięki temu zmniejszamy obciążenie serwera bazy danych i zwiększamy wydajność aplikacji.

Korzystający z obiektu niekoniecznie muszą znać wewnętrzny sposób jego działania. Jeżeli bardziej doświadczony programista w zespole napisze klasę `Widget`, może przekazać ją nowicjuszowi, który, na przykład, nie zna SQL-a, a ten może użyć tego obiektu bez wiedzy, skąd pobierane są dane i w jaki sposób dokonuje się w nich zmian. Można nawet zmienić źródło pochodzenia danych z PostgeSQL na MySQL albo nawet na plik XML bez wiedzy młodego programisty, który nie musi ani o tym wiedzieć, ani modyfikować jakiegokolwiek kodu, w którym wykorzystuje tą klasę.

Ta doskonała technika zostanie zaprezentowana szerzej w rozdziale 7., w którym natrafimy na uogólnioną wersję powyższej klasy, zwaną `GenericObject`, której możemy bez żadnych modyfikacji używać w praktycznie każdym projekcie.

Dziedziczenie

Gdyby zdarzyło się nam tworzyć aplikację obsługującą stan magazynowy w komisie samochodowym, prawdopodobnie potrzebne byłyby nam klasy typu `Sedan`, `Pickup` i `MiniVan`, które odpowiadałyby takim właśnie typom pojazdów będących na stanie komisu. Nasza aplikacja musiałaby nie tylko pokazywać ilość tego rodzaju pojazdów na stanie, ale również charakterystykę każdego z nich, aby handlowcy mogli informować klientów.

Sedan to samochód czterodrzwiowy i najprawdopodobniej zapisać trzeba by liczbę miejsc oraz pojemność bagażnika. Pikap nie posiada bagażnika, ale ma pakę o określonej pojemności, a cały samochód ma pewną dopuszczalną ładowność (maksymalna waga ładunku, jaki może bezpiecznie przewozić). W przypadku samochodu typu minivan przyda się liczba drzwi przesuwnych (jedne lub dwoje) oraz liczba miejsc w środku.

Jednak wszystkie te pojazdy są tak naprawdę tylko pewnymi typami samochodów i jako takie będą miały wiele wspólnych cech w naszej aplikacji, takich jak kolor, producent, model, rok produkcji, numer identyfikacyjny samochodu itp. Aby wszystkie klasy miały te wspólne właściwości, można by po prostu skopiować kod, który je tworzy, do każdego z plików z definicjami klas. Jak wspomniano wcześniej w tym rozdziale, jedną z zalet podejścia obiektowego jest możliwość wielokrotnego wykorzystania kodu, nie musimy więc kopiować kodu, bo możemy ponownie skorzystać z właściwości i metod tych klas w ramach procesu zwanego **dziedziczeniem**. Dziedziczenie to zdolność jednej klasy do przejmowania metod i właściwości klasy nadrzędnej.

Mechanizm dziedziczenia pozwala na zdefiniowanie klasy bazowej, w tym przypadku `Auto➥mobile`, i określenie, że inne klasy są typu `Automobile` i stąd mają takie same właściwości i metody co wszystkie obiekty klasy `Automobile`. Możemy ustalić, że `Sedan` jest klasą typu `Automobile` i dlatego automatycznie dziedziczy wszystko, co zostało zdefiniowane w klasie `Automobile` bez konieczności kopiowania kodu. Potem wystarczy napisać jedynie te właściwości i metody klasy `Sedan`, które nie są wspólne dla wszystkich obiektów klasy `Automobile`. Pozostało więc jedynie zdefiniować różnice — podobieństwa między klasami zostaną odziedziczone po klasie bazowej.

Możliwość wielokrotnego wykorzystania kodu to jedna zaleta dziedziczenia, ale jest też druga, bardzo ważna. Powiedzmy, że mamy klasę `Customer` z metodą `buyAutomobile`. Metoda ta pobiera tylko jeden parametr, obiekt klasy `Automobile`, a w wyniku jej działania wydrukowane zostaną wszystkie papiery dokumentujące transakcję sprzedaży, a samochód zosta-

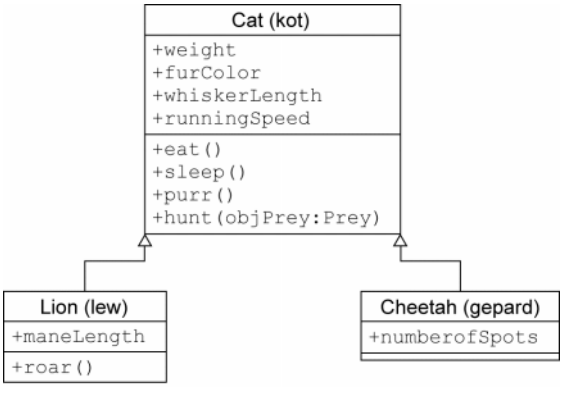
nie usunięty z listy pojazdów na stanie. Ponieważ wszystkie obiekty typu `Sedan`, `Pickup` czy `MiniVan` to obiekty typu `Automobile`, można przesłać każdy z nich do funkcji oczekującej obiektu `Automobile`. Z uwagi na to, że trzy szczególne typy są potomkami bardziej ogólnej klasy nadrzędnej, wiemy, że będą miały ten sam podstawowy zbiór właściwości i metod. Dopóki potrzebne są tylko metody i właściwości wspólne dla wszystkich obiektów `Automob➥ile`, możemy zaakceptować obiekty każdej klasy, która jest potomkiem `Automobile`.

Przeanalizujmy przykład kotów. Wszystkie koty mają pewne wspólne zachowania. Jedzą, śpią, mruczą i polują. Mają też wspólne właściwości — wagę, długość wąsów i szybkość biegania. Z kolei lwy mają grzywy pewnej wielkości (a przynajmniej samce) i ryczą. Gepardy mają cętki. Udomowione koty nie mają żadnej z tych rzeczy, tym niemniej wszystkie wymienione zwierzęta to koty.

W PHP definiujemy klasę jako podzbiór innej klasy, używając słowa kluczowego `extends`, które mówi PHP, że deklarowana właśnie klasa powinna odziedziczyć wszystkie właściwości i metody po swej klasie bazowej oraz że chcemy dodać nowe cechy funkcjonalne lub wyspecjalizować nową klasę.

Gdyby naszym zadaniem było napisanie aplikacji udostępniającej dane o zwierzętach w zoo, prawdopodobnie potrzebowalibyśmy klas `Cat` (kot), `Lion` (lew) i `Cheetah` (gepard). Zanim przejdziemy do pisania kodu, warto zaplanować hierarchię klas za pomocą diagramów UML, aby mieć jakiś punkt wyjścia przy pisaniu kodu i tworzeniu dokumentacji klas (diagramom UML przyjrzymy się dokładniej w rozdziale 2., więc nie obawiajmy się, jeżeli nie do końca rozumiemy to, co tutaj pokazano). Nasz diagram klas powinien ukazywać klasę bazową `Cat` oraz podklasy `Cheetah` i `Lion` będące jej potomkami (patrz rysunek 1.1).

Rysunek 1.1.



Klasy `Lion` i `Cheetah` dziedziczą wszystko po klasie `Cat`, ale klasa `Lion` dodaje implementację właściwości `maneLength` (długość grzywy) oraz metodę `roar()` (ryczenie), a klasa `Cheetah` dodaje właściwość `numberOfSpots` (ilość cętek).

Implementacja klasy `Cat` powinna wyglądać następująco:

```
<?php

   class Cat {
      public $weight;          //w kg
      public $furColor;
```

```
        public $whiskerLength;
        public $maxSpeed;          //w km/h

        public function eat() {
           //kod realizujący jedzenie...
        }

        public function sleep() {
           //kod realizujący spanie...
        }

        public function hunt(Prey $objPrey) {
           //kod realizujący polowanie na obiekty typu Prey (ofiara),
           //których nie będziemy definiować...
        }

        public function purr() {
           print "mrrrrrrr..." . "\n";
        }
     }
  ?>
```

Ta prosta klasa definiuje wszystkie właściwości i metody wspólne wszystkim kotom. Aby stworzyć klasy `Lion` (lew) i `Cheetah` (gepard), można by skopiować cały kod z klasy `Cat` do tych klas. Stwarzamy jednak wtedy dwa problemy. Po pierwsze, jeżeli znajdziemy błąd w klasie `Cat`, będziemy musieli pamiętać o tym, aby poprawić go również w klasach `Lion` i `Cheetah`. I tak oto do zrobienia mamy więcej, a nie mniej (a w końcu zmniejszenie nakładu pracy jest ponoć jedną z podstawowych zalet metody obiektowej).

Po drugie, wyobraźmy sobie, że istnieje metoda jakiegoś innego obiektu, która wygląda następująco:

```
public function petTheKitty(Cat $objCat) {
   $objCat->purr();
}
```

Co prawda głaskanie (`pet...`) lwa lub geparda może nie być rozsądnym pomysłem, ale jeżeli uda się nam podejść na tyle blisko, żeby to zrobić, zapewne zaczną mruczeć (`purr`). Przesłanie obiektu klasy `Lion` i `Cheetah` do metody `petTheKitty()` powinno być możliwe.

Potrzeba więc innego sposobu na stworzenie klas `Lion` i `Cheetah`, a sposobem tym jest właśnie dziedziczenie. Posługując się słowem kluczowym `extends` i określając nazwę klasy „rozszerzanej”, możemy w prosty sposób stworzyć dwie nowe klasy, które mają te same właściwości co klasa `Cat`, uzupełniając je o swoje własne. Na przykład:

```
<?php
   require_once('class.Cat.php');

   class Lion extends Cat {
      public $maneLength;       //w cm

      public function roar() {
         print "Roooooarrrrrr!";
      }
   }
?>
```

I to wszystko! Mając klasę `Lion` rozszerzającą klasę `Cat`, możemy zrobić coś takiego:

```
<?php
   include('class.Lion.php');

   $objLion = new Lion();
   $objLion->weight = 200;     //kg
   $objLion->furColor = 'brąz';
   $objLion->maneLength = 36;  //cm
   $objLion->eat();
   $objLion->roar();
   $objLion->sleep();
?>
```

Możemy więc wywoływać właściwości i metody klasy bazowej `Cat` bez konieczności przepisywania jej kodu. Pamiętajmy, że słowo kluczowe `extends` nakazuje PHP automatyczne dołączenie wszystkich cech funkcjonalnych klasy `Cat` do metod i właściwości specyficznych dla klasy `Lion`. Poza tym informuje PHP, że obiekty klasy `Lion` są również obiektami klasy `Cat` i że możliwe jest wywoływanie funkcji `petTheKitty()` z obiektem klasy `Lion`, nawet jeżeli w deklaracji funkcji użyto nazwy `Cat` jako wskazówki dla parametru:

```
<?php
   include('class.Lion.php');
   $objLion = new Lion();

   $objPetter = new Cat();
   $objPetter->petTheKitty($objLion);
?>
```

W ten sposób wszelkie zmiany dokonane w klasie `Cat` zostają automatycznie uwzględnione w klasie `Lion`. Poprawki, zmiany wewnętrznego sposobu działania funkcji albo nowe metody i właściwości są przesyłane dalej do podklas klasy nadrzędnej. W dużej, dobrze zaprojektowanej hierarchii obiektów może to znacznie ułatwić poprawianie błędów i dodawanie ulepszeń. Drobna zmiana w jednej klasie nadrzędnej, może mieć ogromny wpływ na działanie całej aplikacji.

W kolejnym przykładzie zobaczymy, jak można rozszerzać i specjalizować klasę za pomocą specjalnego konstruktora.

Utwórzmy nowy plik o nazwie *class.Cheetah.php* i wprowadźmy do niego następujący kod:

```
<?php
   require_once('class.Cat.php');

   class Cheetah extends Cat {
      public $numberOfSpots;

      public function __construct() {
         $this->maxSpeed = 100;
      }
   }
?>
```

Poniższy kod wpiszmy w pliku *testcats.php*:

```
<?php
require_once('class.Cheetah.php');

   function petTheKitty(Cat $objCat) {
      if($objCat->maxSpeed < 5) {
         $objCat->purr();
      } else {
         print "Nie da się pogłaskać kotka - oddala się z prędkością " .
               $objCat->maxSpeed . " kilometrów na godzinę!";
      }
   }

   $objCheetah = new Cheetah();
   petTheKitty($objCheetah);

   $objCat = new Cat();
   petTheKitty($objCat);
?>
```

Klasa `Cheetah` dodaje nową publiczną zmienną składową `numberOfSpots` (ilość cętek) oraz konstruktor, który nie występował w nadrzędnej klasie `Cat`. Teraz, kiedy stworzymy nowy obiekt `Cheetah`, właściwość `maxSpeed` (odziedziczona po klasie `Cat`) zostanie zainicjalizowana wartością 100 kilometrów na godzinę, co jest w przybliżeniu maksymalną prędkością osiąganą przez gepardy na krótkich dystansach. Zauważmy, że niepodanie wartości domyślnej dla klasy `Cat` sprawia, że wartość zmiennej `maxSpeed` dla funkcji `petTheKitty()` wynosi 0 (a dokładnie wartość pusta, czyli `null`). Jak wiedzą ci, którzy kiedykolwiek mieli kota domowego, czas jaki ten poświęca na spanie, sprawia, że jego maksymalna prędkość jest bliska zeru!

Dodając nowe funkcje, właściwości, a nawet konstruktory i destruktory, podklasy klasy nadrzędnej mogą łatwo rozszerzać swoje możliwości funkcjonalne i tym samym minimalną ilością kodu uzupełniać aplikację o nowe elementy.

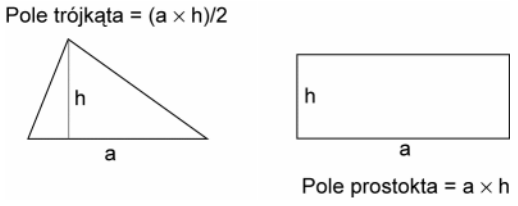
> Jeżeli można powiedzieć, że jedna klasa jest szczególnym typem innej klasy, to należy wykorzystać dziedziczenie w celu maksymalnego skorzystania z potencjalnej możliwości wielokrotnego wykorzystania kodu i uelastycznienia aplikacji.

Zastępowanie metod

To, że klasa jest potomkiem innej klasy, nie oznacza, iż zawsze musi korzystać z implementacji funkcji pochodzącej od przodka. Gdybyśmy pisali aplikację, która oblicza pola różnych figur geometrycznych, mogłyby pojawić się w niej klasy `Rectangle` (prostokąt) i `Triangle` (trójkąt). Obydwie figury są wielokątami i jako takie będą potomkami klasy `Polygon` (wielokąt).

Klasa `Polygon` będzie miała właściwość `numberOfSides` (ilość boków) oraz metodę `getArea` (zwróć pole). Obliczenie pola jest możliwe dla każdego wielokąta, ale metoda jego obliczania będzie dla każdego wielokąta inna (istnieje ogólne równanie pola wielokąta, ale często okazuje się mniej wydajne niż równania dla konktretnych figur — w tym przypadku prostych wielokątów). Wzór na pole prostokąta to `a * h`, gdzie `a` jest szerokością prostokąta, a `h` jego wysokością. Pole trójkąta to `0,5 * a * h`, gdzie `a` to jego podstawa, a `h` to wysokość. Rysunek 1.2 przedstawia niektóre przykłady obliczania pól różnych wielokątów.

Rysunek 1.2.



Dla każdej utworzonej podklasy klasy `Polygon` będziemy prawdopodobnie chcieli użyć równania innego niż w domyślnej implementacji metody obliczającej pole posługującej się równaniem specyficznym dla ogólnych wielokątów. Poprzez redefinicję metody klasy możemy wprowadzić własną wersję implementacji.

W przypadku klasy `Rectangle` stworzylibyśmy dwie nowe właściwości `height` (wysokość) i `width` (szerokość) i zastąpili implementację `getArea()` pochodzącą z klasy `Polygon` inną wersją. W przypadku klasy `Triangle` dodalibyśmy zapewne właściwości przechowujące dane trzech kątów, długość odcinka podstawy i wysokość trójkąta i także zastąpili metodę `getArea()`. Posługując się dziedziczeniem i zastępowaniem metod klas nadrzędnych, możemy tworzyć wyspecjalizowane implementacje tych metod na poziomie klas potomnych.

Funkcja, która pobiera `Polygon` jako parametr i wyświetla pole tego wielokąta, będzie wówczas automatycznie wywoływać metodę `getArea()` przynależną podklasie klasy `Polygon` zgodnej z typem obiektu do niej przesłanego (czyli np. `Rectangle` albo `Triangle`). Zdolność języka obiektowego do automatycznego określania w trakcie wykonywania programu, która metoda `getArea()` ma zostać wywołana, nazywana jest **polimorfizmem**. Polimorfizm to zdolność aplikacji do robienia różnych rzeczy w zależności od obiektu odniesienia. W tym przypadku sprowadza się to do wywoływania różnych wersji metod `getArea()`.

> Metody w podklasach należy zastępować, kiedy implementacja pochodząca z klasy nadrzędnej różni się od tej wymaganej w podklasie. Pozwala to na wyspecjalizowanie operacji danej podklasy.

Czasami chcemy zachować implementację pochodzącą z klasy nadrzędnej, wykonując jedynie pewne dodatkowe czynności w metodzie podklasy. W aplikacji, która wspomaga zarządzanie organizacją charytatywną, prawdopodobnie występowałaby klasa `Volunteer` (wolontariusz) z metodą `signUp()`, która umożliwiałaby wolontariuszowi na przystąpienie do jednej z organizowanych akcji i dopisywałaby go do listy osób, które zgłosiły chęć uczestnictwa.

Załóżmy, że niektórym użytkownikom przypisane są pewne ograniczenia, takie jak bycie karanym, zakazujące im uczestnictwa w pewnych przedsięwzięciach. W takim przypadku polimorfizm umożliwia stworzenie klasy `RestrictedUser` (użytkownik o ograniczonych prawach) wprowadzającej własną wersję metody `signUp()`, która w pierwszym kroku konfrontuje ograniczenia na koncie użytkownika z właściwościami danego przedsięwzięcia i uniemożliwia przystąpienie do niego, jeżeli ograniczenia zakazują uczestnictwa w pewnego rodzaju akcjach. Jeżeli ograniczenia nie uniemożliwiają uczestnictwa, należy wywołać odpowiednie metody klasy nadrzędnej, które pozwolą dopełnić rejestracji.

Zastępując metodę klasy nadrzędnej, nie trzeba pisać jej zupełnie od nowa. Można wciąż korzystać z implementacji pochodzącej od przodka, uzupełniając ją o pewne specjalizujące elementy na poziomie podklasy. W taki sposób możemy wielokrotnie wykorzystywać kod i dostosowywać go do wymagań obszaru zastosowania.

Zdolność jednej klasy do dziedziczenia metod i właściwości innej klasy jest jedną z najbardziej pociągających cech systemów obiektowych, pozwalając na uzyskanie niesamowicie wysokiego poziomu efektywności i elastyczności aplikacji.

W naszym przykładzie stworzymy dwie klasy — `Rectangle` (prostokąt) i `Square` (kwadrat). Kwadrat to szczególny przypadek prostokąta. Wszystko, co można zrobić z prostokątem, można również zrobić z kwadratem, ale z uwagi na to, że w prostokącie występują dwie długości boków, a w kwadracie tylko jedna, niektóre operacje trzeba wykonać inaczej.

Stwórzmy plik *class.Rectangle.php* i dodajmy do niego następujący kod:

```
<?php

   class Rectangle {
      public $height;
      public $width;

      public function __construct($width, $height) {
         $this->width = $width;
         $this->height = $height;
      }

      public function getArea() {
        return $this->height * $this->width;
      }
   }
?>
```

Jest to w miarę oczywista implementacja klasy modelującej prostokąt. Konstruktor pobiera szerokość i wysokość jako parametry, a funkcja `getArea()` oblicza pole prostokąta, mnożąc te wartości przez siebie.

Spójrzmy teraz na treść klasy opisującej kwadrat (*class.Square.php*):

```
<?php
   require_once('class.Rectangle.php');

   class Square extends Rectangle {
      public function __construct($size) {
         $this->height = $size;
         $this->width = $size;
      }

      public function getArea() {
         return pow($this->height, 2);
      }

   }
?>
```

W kodzie tym zastąpiony został zarówno konstruktor, jak i metoda `getArea()`. Aby kwadrat mógł być kwadratem, wszystkie boki muszą być tej samej długości. W efekcie konstruktor potrzebuje tylko jednego parametru. Jeżeli do funkcji przesłanych zostanie więcej parametrów, to wszystkie oprócz pierwszego zostaną zignorowane.

PHP nie generuje błędu, jeżeli liczba parametrów przekazanych do funkcji zdefiniowanej przez użytkownika jest większa niż liczba ustalona w treści deklaracji. W wielu przypadkach takie zachowanie jest pożądane. Więcej na ten temat można wyczytać z dokumentacji wbudowanej funkcji `func_get_args()`.

Funkcja `getArea()` również została zastąpiona. Implementacja w klasie `Rectangle` dawałaby co prawda prawidłowe wyniki dla obiektów `Square`, ale została zastąpiona w celu poprawienia wydajności aplikacji (chociaż w tym akurat przypadku poprawa jest symboliczna). W PHP pobranie jednej wartości właściwości i podniesienie jej do kwadratu trwa krócej niż pobranie dwóch wartości i ich pomnożenie.

Zastępując konstruktory, destruktory i metody, możemy modyfikować różne aspekty funkcjonowania podklas.

Zachowywanie możliwości funkcjonalnych przodka

Czasami chcemy zachować pewne możliwości funkcjonalne odziedziczone po przodku. Funkcji nie potrzeba zawsze zastępować w całości — można jedynie coś do nich dodać. Można by skopiować kod z metody przodka, ale jak już widzieliśmy, metoda obiektowa daje nieco lepsze możliwości niż kopiowanie kodu.

Aby wywołać możliwości funkcjonalne zaimplementowane u przodka, należy posłużyć się składnią `parent::[`*`nazwa funkcji`*`]`. Kiedy chcemy jedynie uzupełnić metodę przodka o pewne zachowania, wystarczy wywołać `parent::[`*`nazwa funkcji`*`]`, po czym dodać uzupełniający kod. Rozbudowując funkcję w ten sposób, zawsze na początku należy wywołać metodę przodka. W ten sposób sprawimy, że wszelkie zmiany implementacji przodka nie zaburzą działania kodu u potomka.

Ponieważ klasa przodka może oczekiwać pewnego stanu od obiektu albo zmieniać go, modyfikować wartości właściwości albo manipulować wewnętrznymi danymi obiektu, należy podczas rozbudowy metody odziedziczonej zawsze wywoływać metodę przodka przed kodem rozbudowującym.

W poniższym przykładzie występują dwie klasy: `Customer` (klient) i `SweepstakesCustomer` (klient w czasie promocji). W supermarkecie działa aplikacja, która w czasie trwania pewnych promocji zamienia klasy używane przez aplikację zarządzającą kasami. Każdy klient przechodzący przez kasę ma swój własny numer identyfikacyjny (który pochodzi z bazy danych) oraz numer klienta, który wskazuje, ilu klientów przeszło przed nim przez kasę. Promocja polega na tym, że milionowy klient wygrywa nagrodę.

Stwórzmy plik o nazwie *class.Customer.php* i wpiszmy w nim następujący kod:

```
<?php

   class Customer {
      public $id;
      public $customerNumber;
      public $name;

      public function __construct($customerID) {
         //pobiera dane klienta z bazy
         //
```

```
            //Tutaj akurat wartości wpisano na stałe,
            //ale normalnie powinny pochodzić z bazy danych
            $data = array();
            $data['customerNumber'] = 1000000;
            $data['name'] = 'Janina Morszczuk';

            //Przypisuje wartości z bazy danych do obiektu
            $this->id = $customerID;
            $this->name = $data['name'];
            $this->customerNumber = $data['customerNumber'];
        }
    }
?>
```

Utwórzmy plik o nazwie *class.SweepstakesCustomer.php* i wpiszmy w nim następujący kod:

```
<?php
   require_once('class.Customer.php');

   class SweepstakesCustomer extends Customer {
      public function __construct($customerID) {
         parent::__construct($customerID);

         if($this->customerNumber == 1000000) {
            print "Gratulacje $this->name! Jesteś naszą milionową klientką! " .
                  "Wygrałaś roczny zapas mrożonych filetów z morszczuka! ";
         }
      }
   }

?>
```

Jak działa dziedziczenie?

Klasa `Customer` inicjalizuje wartości zmiennych danymi z bazy na podstawie identyfikatora klienta. Taki numer najczęściej można uzyskać, odczytując dane z karty programu lojalnościowego wydawanej przez większość sieci supermarketów. Mając identyfikator klienta, możemy pobrać dane osobowe klienta z bazy (w tym przykładzie zostały zapisane na stałe w kodzie) oraz liczbę reprezentującą ilość klientów, jaka weszła do sklepu przed tym klientem. Wszystkie te informacje zostają zachowane w publicznych zmiennych składowych.

Klasa `SweepstakesCustomer` dodaje pewną możliwość funkcjonalną do konstruktora. Na początku wywoływany jest kod konstruktora pochodzący od przodka poprzez `parent::__construct` i przesyłane są do niego oczekiwane parametry.. Następnie sprawdzana jest wartość właściwości `customerNumber`. Jeżeli dany klient jest klientem milionowym, to generowany jest komunikat o wygranej.

Aby przetestować działanie tej klasy, stwórzmy plik o nazwie *testCustomer.php* i wpiszmy w nim następujący kod:

```
<?php

   require_once('class.SweepstakesCustomer.php');
   //ponieważ plik ten zawiera już w sobie class.Customer.php,
```

```
    //nie ma potrzeby dodatkowego dołączania tego pliku.

    function greetCustomer(Customer $objCust) {
       print "$objCust->name, witamy Cię ponownie w naszym sklepie!";
    }

    //Zmiana tej wartości powoduje zmianę klasy, z której stworzony zostanie obiekt klienta
    $promotionCurrentlyRunning = true;

    if ($promotionCurrentlyRunning) {
       $objCust = new SweepstakesCustomer(12345);
    } else {
       $objCust = new Customer(12345);
    }

    greetCustomer($objCust);

 ?>
```

Teraz możemy uruchomić *testCustomer.php* w przeglądarce ze zmienną `$promotionCurrent➥lyRunning` ustawioną najpierw na `false`, a potem na `true`. Kiedy ustawimy tą wartość na `true`, ukaże się komunikat o wygranej.

Interfejsy

Czasami mamy grupy klas powiązane relacjami niekoniecznie polegającymi na dziedziczeniu. Zdarza się, że zupełnie odmienne klasy mają pewne wspólne zachowania. Na przykład zarówno słoik, jak i drzwi można otworzyć i zamknąć, mimo że poza tym faktem nie mają ze sobą nic wspólnego. Niezależnie od rodzaju słoika i drzwi obydwu tym rzeczom można przypisać te same operacje, ale poza tym nic ich nie łączy.

Co robią interfejsy?

Taka sama idea jest obecna w programowaniu obiektowym. **Interfejs** pozwala na określenie, że obiekt jest w stanie wykonać pewną funkcję bez konieczności tłumaczenia, w jaki sposób się to odbywa. Interfejs to swoista umowa pomiędzy niepowiązanymi obiektami zawiązana w celu wykonania wspólnej funkcji. Obiekt, który implementuje dany interfejs, gwarantuje użytkownikom, że potrafi wykonać wszystkie funkcje wymienione w specyfikacji interfejsu. Rowery i piłki to zupełnie różne rzeczy, a mimo to obiekty reprezentujące te rzeczy w oprogramowaniu sklepu sportowego muszą prowadzić interakcję z tym systemem.

Deklarując interfejs, po czym implementując go w obiektach, możemy umożliwić zupełnie różnym klasom dostęp do wspólnych funkcji. Przedstawiony tu przykład opiera się na raczej prozaicznej analogii do drzwi i słoika.

Stwórzmy plik o nazwie *interface.Opener.php*:

```
 <?php

    interface Openable {
       abstract function open();
```

```
        abstract function close();
    }

?>
```

Analogicznie do konwencji nazywania plików z klasami wg wzorca *class.[Nazwa klasy].php* powinniśmy zastosować coś podobnego dla plików interfejsów i nazywać je wg wzorca *inter➥face.[Nazwa interfejsu].php*.

Deklarujemy interfejs `openable`, posługując się składnią podobną do składni klasy; poza tym, że słowo `class` zastępujemy słowem `interface`. Interfejs nie posiada zmiennych składowych i nie precyzuje implementacji swoich funkcji składowych.

Ponieważ nie podaje się tu żadnej implementacji, funkcje są deklarowane jako abstrakcyjne (`abstract`). Informuje to PHP, że każda klasa implementująca ten interfejs jest odpowiedzialna za dostarczenie implementacji występujących w nim funkcji. Jeżeli nie zostaną dostarczone implementacje *wszystkich* abstrakcyjnych metod interfejsu, PHP wygeneruje błąd po uruchomieniu programu. Nie można wybiórczo implementować niektórych metod abstrakcyjnych — konieczne jest zaimplementowanie wszystkich.

Jak działają interfejsy?

Interfejs `openable` to umowa z innymi częściami aplikacji mówiąca, że każda klasa implementująca ten interfejs dostarczy dwóch metod, zwanych `open()` i `close()`, które nie pobierają parametrów. Mając uzgodniony zestaw metod, możemy sprawić, by zupełnie różne obiekty były przekazywane do tych samych funkcji bez potrzeby tworzenia między nimi relacji opartych na dziedziczeniu.

Utwórzmy kilka plików. Zacznijmy od *class.Door.php*:

```
<?php

require_once('interface.Openable.php');

class Door implements Openable {

   private $_locked = false;

   public function open() {
      if($this->_locked) {
         print "Nie można otworzyć drzwi. Są zamknięte na klucz.";
      } else {
        print "skrzyyyyp...<br>";
      }
   }

   public function close() {
      print "Trrrrach!!<br>";
   }

   public function lockDoor() {
      $this->_locked = true;
   }
```

```
    public function unlockDoor() {
       $this->_locked = false;
    }

 }
```

następnie *class.Jar.php*:

```
<?php
require_once('interface.Openable.php');

class Jar implements Openable {
   private $contents;

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

   public function open() {
      print "słoik został otwarty<br>";
   }

   public function close() {
      print "słoik został zamknięty<br>";
   }
}
?>
```

Aby skorzystać z tych plików, stworzymy w tym samym katalogu jeszcze jeden plik o nazwie *testOpenable.php*:

```
<?php
   require_once('class.Door.php');
   require_once('class.Jar.php');

   function openSomething(Openable $obj) {
      $obj->open();
   }

   $objDoor = new Door();
   $objJar = new Jar("galaretka");

   openSomething($objDoor);
   openSomething($objJar);
?>
```

Ponieważ zarówno klasa `Door`, jak i `Jar` implementują interfejs `Openable`, można przesłać obiekty każdej z tych klas do funkcji `openSomething()`. Ponieważ funkcja ta akceptuje tylko coś, co implementuje interfejs `Openable`, wiemy, że możemy w jej ramach wywołać funkcje `open()` i `close()`. Nie należy jednak próbować dostępu do właściwości `contents` (zawartość) słoika ani używać funkcji `lock()` lub `unlock()` otwierających zamek drzwi w klasie `Door` z poziomu funkcji `openSomething()`, ponieważ ta właściwość i te metody nie są częścią interfejsu. Zgodnie z umową interfejsu możemy jedynie otwierać funkcją `open()` i zamykać funkcją `close()`.

Stosując interfejsy w aplikacji, możemy urzeczywistnić komunikację pomiędzy dwoma zupełnie niepowiązanymi z sobą obiektami z gwarancją, że ich interakcja będzie odbywać się w ramach warunków określonych w interfejsie. Interfejs jest umową udostępniającą pewne metody.

Hermetyzacja

Jak wspomniano wcześniej w tym rozdziale, obiekty pozwalają na ukrycie szczegółów ich implementacji przed ich użytkownikami. Nie musimy wiedzieć, czy wspomniana wcześniej klasa `Volunteer` (wolontariusz), zachowuje informacje w bazie danych, w zwykłym pliku tekstowym czy w dokumencie XML lub innym mechanizmie przechowywania danych, aby móc wywołać metodę `signUp()`. Podobnie nie musimy wiedzieć, czy dane wolontariusza przechowywane w obiekcie są reprezentowane jako pojedyncze zmienne, tablica czy może inny obiekt. Zdolność do ukrywania implementacji to **hermetyzacja**. Ogólnie rzecz biorąc, hermetyzacja ma dwa aspekty — ochrona wewnętrznych danych klasy przed zewnętrznym kodem oraz ukrywanie szczegółów implementacji.

Słowo *encapsulate* oznaczające hermetyzowanie w języku angielskim oznacza dosłownie umieszczenie w kapsule lub w innym wyodrębnionym pojemniku. Dobrze zaprojektowana klasa, buduje szczelną barierę wokół swego wnętrza i udostępnia dla zewnętrznego kodu interfejs, który jest całkowicie odseparowany od tego wnętrza. Ma to dwie zalety: możemy w każdej chwili zmieniać szczegóły implementacji, nie wywierając wypływu na działanie kodu korzystającego z klasy, a także mamy pewność, że nic spoza klasy nie może niepostrzeżenie zmodyfikować wartości określających stan i właściwości obiektu zbudowanego z klasy i tym samym ufać, że stan obiektu i wartości właściwości będą prawidłowe.

Zmienne i funkcje składowe klasy mają określoną widzialność. **Widzialność** odnosi się do tego, co jest widoczne dla kodu spoza klasy. **Prywatne** funkcje i zmienne składowe nie są dostępne dla kodu zewnętrznego i służą potrzebom wewnętrznej implementacji klasy. Składowe **chronione** są widoczne tylko z poziomu podklas danej klasy. Składowe **publiczne** mogą być wykorzystywane z poziomu każdego kodu, z wnętrza i spoza klasy.

Ogólnie rzecz biorąc, wszystkie wewnętrzne składowe klasy powinny być deklarowane jako prywatne. Każdy dostęp do tych zmiennych dla kodu spoza klasy powinien być realizowany poprzez metody dostępowe. Nikt z nas nie zgodziłby się zapewne na degustację nowej potrawy z zawiązanymi oczami i przez karmienie na siłę. Wolelibyśmy przyjrzeć się daniu i zadecydować, czy naprawdę chcemy je zjeść. Podobnie, kiedy obiekt chce dopuścić możliwość zmiany swoich właściwości lub innego rodzaju wpływu na wewnętrzne dane z poziomu zewnętrznego kodu, to dzięki hermetyzacji dostępu do tych danych za pośrednictwem funkcji publicznych (i zachowaniu prywatności wewnętrznych danych) zyskujemy możliwość weryfikacji zmian i ich akceptacji bądź odrzucenia.

Jeżeli, na przykład, tworzymy aplikację dla banku, która obsługuje szczegóły związane z rachunkami klientów, to być może będziemy mieli obiekt `Account` (rachunek) z właściwością `totalBalance` (saldo końcowe) oraz metodą `makeDeposit()` realizującą wpłatę i metodą `make➥Withdrawal()` realizującą wypłatę. Właściwość reprezentująca bilans powinna być tylko do odczytu. Jedynym sposobem na zmianę salda jest dokonanie wpłaty lub wypłaty. Gdyby zaimplementować właściwość `totalBalance` jako składową publiczną, to możliwe byłoby napisanie kodu, który zwiększałby wartość tej zmiennej bez konieczności dokonywania wpłaty.

Takie podejście nie byłoby zbyt dobre dla banku. Lepiej więc zaimplementować tę właściwość jako prywatną zmienną składową i udostępnić publiczną metodę zwaną `getTotalBala➥nce()`, która zwraca wartość prywatnej zmiennej składowej. Ponieważ zmienna przechowująca wartość salda rachunku jest prywatna, nie można bezpośrednią nią manipulować. Z uwagi na to, że jedynymi publicznymi metodami, które mają wpływ na saldo rachunku, są `makeWithdrawal()` i `makeDeposit()`, zwiększenie salda rachunku będzie wymagało dokonania wpłaty.

Umożliwiając ukrycie szczegółów implementacji i ochronę dostępu do wewnętrznych zmiennych składowych, programowanie obiektowe pozwala na tworzenie stabilnych i elastycznych aplikacji.

> Hermetyzacja danych wewnętrznych i implementacji metod umożliwia systemowi oprogramowania obiektowego ochronę i kontrolę dostępu do danych oraz ukrywanie szczegółów implementacji.

Zmiany w PHP5 dotyczące programowania obiektowego

Obiekty były obsługiwane w PHP już od wersji PHP3. Nie wiązało się to jednak z intencją obsługi idei klas i obiektów — dodano jedynie pewną ograniczoną obsługę, bardziej w formie dodatku dostarczającego „składniową osłodę" (posługując się słowami Zeeva Suraskiego) tablicom asocjacyjnym. Obsługa obiektów w PHP pierwotnie stanowiła wygodny sposób grupowania danych i funkcji, ale uwzględniała tylko niewielki podzbiór cech funkcjonalnych, jakie posiadały języki w pełni obiektowe. W miarę wzrostu popularności PHP stosowanie podejścia obiektowego stawało się coraz powszechniejsze w dużych aplikacjach. Słaba implementacja obiektowości zaczęła krępować ruchy.

Co najważniejsze, nie było żadnej obsługi prawdziwej hermetyzacji. Nie można było opisywać zmiennych czy metod jako prywatnych lub chronionych. Wszystko było publiczne, co, jak widzieliśmy, bywa źródłem problemów.

Nie było też obsługi abstrakcyjnych interfejsów i metod. Metody i zmienne składowe nie mogły być deklarowane jako statyczne. Nie istniały destruktory. Wszystkie te pojęcia były znane każdemu, kto miał do czynienia z innym językiem obiektowym i brak tych możliwości w modelu obiektowym PHP utrudniał przejście do PHP z takich języków jak Java (która obsługuje wszystkie te aspekty). Ci, którzy mieli wcześniej doświadczenia z PHP4, mogą z poniższej tabeli wyczytać nowe cechy modelu obiektowego wprowadzone w PHP5.

Nowa cecha	Korzyści
Prywatne i chronione zmienne i funkcje składowe.	Od tej chwili również PHP umożliwia pełną hermetyzacje i ochronę danych.
Ulepszona obsługa usuwania pośredniości.	Można stosować takie instrukcje jak `$obj->getSomething()->doSomething()`.

Nowa cecha	Korzyści
Zmienne i metody składowe mogą być statyczne.	Metody, które mogą być wywoływane statycznie, są teraz jasno identyfikowalne. Stałe na poziomie klasy pomagają ograniczać zanieczyszczenie globalnej przestrzeni nazw.
Jednorodna postać konstruktorów.	Konstruktory klasy oznaczamy teraz jako `__construct()`. Pomaga to w hermetyzacji zastąpionych konstruktorów podklas i ułatwia zmianę dziedziczenia, kiedy struktura dziedziczenia składa się z wielu klas.
Obsługa destruktorów.	Dzięki metodzie `__destruct()` klasy w PHP mogą mieć już destruktory. Umożliwiają one wykonywanie określonych czynności w chwili likwidacji obiektu.
Obsługa abstrakcyjnych klas i interfejsów.	Można definiować potrzebne metody w klasie przodka, wstrzymując się z implementacją i wprowadzając ją dopiero w podklasie. Nie można tworzyć egzemplarzy klas abstrakcyjnych, tylko ich skonkretyzowanych podklas.
Sugestie typów parametrów.	Można wskazać klasę dla tych parametrów funkcji, które oczekują obiektu. Pisząc `function foo(Bar $objBar) {...`, mamy pewność, że typ parametru jest zgodny z naszymi oczekiwaniami

Podsumowanie

W rozdziale omówiliśmy koncepcję programowania obiektowego. Opisaliśmy klasę jako wzorzec dla tworzenia obiektów. Obiekty to istniejące w czasie egzekucji zbiory funkcji i danych stworzone na podstawie definicji klas. Obiekty mają cechy charakterystyczne zwane **właściwościami** i zachowania zwane **metodami**. Właściwości można postrzegać jako zmienne, a metody jako funkcje.

Niektóre klasy mają wspólnego przodka. Kwadraty są prostokątami. Kiedy deklarujemy klasę jako podtyp klasy nadrzędnej, dziedziczy ona metody i właściwości klasy nadrzędnej. Możliwe jest zastępowanie metod odziedziczonych. Można napisać zupełnie nową implementację lub zdecydować się na użycie implementacji pochodzącej od przodka i dodanie do niej kodu specjalizującego charakterystycznego dla danej podklasy. Można też w ogóle nie zastępować metody.

Hermetyzacja to ważne pojęcie w programowaniu obiektowym. Oznacza ono zdolność klasy do ochrony dostępu do swoich wewnętrznych zmiennych składowych i odgrodzenia użytkowników klasy od szczegółów implementacyjnych. Metody i właściwości mają trzy poziomy widzialności — prywatny, chroniony i publiczny. Składowe prywatne mogą być używane wyłącznie przez wewnętrzne operacje klasy. Składowe chronione są widoczne z poziomu podklas. Składowe publiczne mogą być używane przez kod spoza klasy.

Obsługa programowania obiektowego w PHP uległa poważnej modernizacji w PHP5 i module Zend Engine 2. Nowe cechy i znaczne zwiększenie wydajności czynią PHP językiem obiektowym w pełnym tego słowa znaczeniu.