helion.pl

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

Zaawansowane programowanie w PHP4

Autor: praca zbiorowa
ISBN: 83-7197-729-8
Tytuł oryginału: Professional PHP 4
Format: B5, stron: 990

W opinii autorów niniejsza książka stanowi doskonałe źródło wiedzy dla aktywnych programistów, wykorzystujących w swojej pracy PHP.
Ta książka ma na celu umożliwienie twórcom aplikacji internetowych tworzenie programów, które będą:

- skalowalne,
- wydajne,
- bezpieczne,
- modularne,
- wielowarstwowe.

Książka ta adresowana jest do programistów, których znajomość PHP pozwala na tworzenie i rozwijanie niewielkich aplikacji WWW. Pomimo że przedstawiamy składnię PHP, liczymy na to, że programiści czytający tę książkę nie będą potrzebowali wykładu na temat podstaw programowania.

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

Spis treści

O Autorach 19

Wstęp 23
- Dla kogo jest przeznaczona ta książka? 23
- Zawartość książki 24
- Czego potrzeba, aby można było korzystać z tej książki? 27
- Konwencje 27

Rozdział 1. Droga do PHP 29
- Dlaczego PHP? 29
- Ewolucja języka PHP 30
  - Przeszłość PHP 30
  - PHP dzisiaj 30
  - PHP na ringu 30
  - Przyszłość PHP 31
- PHP a inne języki 31
  - PHP a ASP 31
  - PHP a Cold Fusion 32
  - PHP a Perl 32
  - PHP a Java 32
- Licencje PHP 33
- Dodatkowe źródła informacji 33

Rozdział 2. Instalacja 35
- Mamy już PHP 35
- Przed instalacją 37
- Decyzje instalacyjne 38
  - Który system operacyjny? 38
  - Moduł czy CGI? 38
  - Który serwer WWW? 40
- Instalacja MySQL, Apache i PHP 40
- Instalacja w systemie Windows 40
  - Instalacja bazy MySQL 41
    - Jakie komplikacje mogą wystąpić? 42
  - Instalacja serwera Apache 43
    - Jakie komplikacje mogą wystąpić? 46
  - Instalacja PHP 46
  - Konfiguracja Apache do obsługi PHP 48
    - Jakie komplikacje mogą wystąpić? 49

Testowanie instalacji PHP ... 50
Jakie komplikacje mogą wystąpić? ... 51
Czynności po instalacji ... 52
Przejście na ISAPI ... 54
Instalacja w systemach klasy UNIX ... 54
Instalacja MySQL ... 55
Utworzenie użytkownika dla MySQL-a ... 55
Konfiguracja kodu źródłowego MySQL-a ... 56
Kompilacja MySQL-a ... 57
Inicjalizacja MySQL-a ... 58
Uruchamianie MySQL-a ... 59
Testowanie MySQL-a ... 59
Zabezpieczanie MySQL-a ... 60
Jakie komplikacje mogą wystąpić? ... 60
Instalacja Apache ... 60
Czynności po instalacji Apache ... 61
Jakie komplikacje mogą wystąpić? ... 62
Instalacja PHP ... 63
Jakie komplikacje mogą wystąpić? ... 64
Kompilacja PHP ... 65
Jakie komplikacje mogą wystąpić? ... 66
Czynności po instalacji ... 66
Integracja PHP z Apache ... 66
Jakie komplikacje mogą wystąpić? ... 68
Czynności po instalacji ... 69
Dodatkowe źródła informacji ... 71
PHP.net ... 71
Zend.com ... 72
php4win.de ... 72
Apache ... 73
MySQL ... 73

Rozdział 3. Podstawy PHP ... 75

Programy w PHP ... 75
Skrypty PHP ... 76
Instrukcje ... 77
Komentarze ... 79
Literały ... 80
Literały tekstowe ... 80
Dokumenty osadzone ... 81
Literały liczbowe ... 82
Literały logiczne ... 82
Zmienne ... 83
Przypisanie ... 83
Odwołanie ... 84
Stałe ... 85
Typy danych ... 85
Rzutowanie ... 86
Operatory i funkcje ... 87
Operacje ogólnego przeznaczenia ... 89
Operacje na napisach ... 90
Funkcje napisowe ... 91
substr() ... 91
strpos() ... 91

htmspecialchars() ... 92
trim() ... 92
chr() oraz ord() ... 93
strlen() ... 93
printf() oraz sprintf() ... 93
Operacje liczbowe ... 95
Operacje bitowe ... 96
Operacje porównania ... 97
Priorytety operatorów ... 98
Operacje logiczne ... 98
Priorytety operatorów ... 99
Tablice ... 99
Zmienne zewnętrzne ... 99
Zmienne systemowe, zmienne GET oraz $HTTP_ Arrays ... 100
Zmienne POST ... 101
Cookies ... 101
Zmienne CGI ... 101
Zmienne nagłówków HTTP ... 102

Rozdział 4. Struktury w PHP ... 103

Struktury kontroli przebiegu programu ... 103
Instrukcje warunkowe ... 103
If ... 103
switch ... 106
Pętle ... 107
while ... 108
do...while ... 109
for ... 109
Alternatywna składnia pętli ... 110
Funkcje ... 110
Definiowanie funkcji ... 110
Zakres zmiennej ... 112
Czas życia zmiennej ... 113
Rekurencja ... 114
Przypisywanie funkcji zmiennym ... 114
Zastosowanie funkcji w celu uporządkowania kodu ... 115
Komentarze ... 118
Tablice ... 118
Inicjacja tablic ... 119
Sekwencyjne przeglądanie tablic ... 120
Wbudowane funkcje tablicowe ... 120
count() ... 120
in_array() ... 121
reset() ... 121
sort() ... 121
explode() oraz implode() ... 122
Tablice predefiniowane ... 122
Tablice wielowymiarowe ... 122

Rozdział 5. Programowanie obiektowe w PHP4 ... 125

Programowanie zorientowane obiektowo ... 125
Programowanie proceduralne a programowanie obiektowe ... 127
Znaczenie programowania obiektowego ... 128

Zstępująca metoda tworzenia oprogramowania ... 128
Klasy ... 129
Obiekty ... 132
Metody fabryczne ... 133
Hermetyzacja (ang.Encapsulation) ... 135
Dziedziczenie ... 137
Operator wywołania metody klasy ... 141
Ponowne wykorzystanie kodu ... 141
Polimorfizm ... 142
Metody abstrakcyjne ... 143
Adekwatność i powiązania ... 146
Modelowanie obiektowe z użyciem UML ... 148
Delegacja ... 150
Analiza i decyzje projektowe ... 152
Funkcje PHP obsługujące klasy ... 154
get_class() ... 154
get_parent_class() ... 155
Ograniczenia PHP ... 155
Brak atrybutów statycznych ... 156
Brak destruktorów ... 157
Brak wielokrotnego dziedziczenia ... 158
Modelowanie złożonego komponentu WWW ... 160

Rozdział 6. Wykrywanie i usuwanie błędów ... 165

Przegląd błędów programistycznych ... 166
Błędy składni ... 166
Błędy semantyczne ... 167
Błędy logiczne ... 168
Błędy środowiska ... 169
Poziomy błędów w PHP ... 169
Błędy analizy ... 170
Błędy krytyczne ... 170
Ostrzeżenia ... 170
Uwagi ... 170
Błędy na poziomie jądra ... 171
Poziomy błędów etapu kompilacji ... 171
Poziomy błędów definiowanych przez użytkownika ... 171
Ustawianie poziomów zgłaszania błędów ... 171
Obsługa błędów ... 172
Wyciszanie komunikatów błędach ... 172
Postępowanie w przypadku wystąpienia błędu ... 173
Sprawdzanie błędów w nietypowych sytuacjach ... 174
Raportowanie błędów ... 175
Programy wspomagające wykrywanie błędów ... 176
Narzędzia do wykrywania błędów wykorzystujące protokół HTTP ... 177
Klient telnet ... 177
Serwery nasłuchujące ... 178
Metoda śledzenia ... 179
phpCodeSite ... 180
Zdalne systemy wykrywania błędów ... 185
BODY ... 185
Zend IDE ... 187
Testowanie skryptowe ... 188

Rozdział 7. Wprowadzanie danych i wyrażenia regularne ... 193
Wprowadzanie danych ... 193
Formularze ... 194
Formularze HTML ... 194
Atrybut action ... 195
Atrybut method ... 195
Obsługa wprowadzanych danych ... 196
Skomplikowane formularze ... 197
Weryfikacja danych ... 200
OOH Forms ... 201
Przykładowa aplikacja ... 201
Zabezpieczenie przed niewłaściwym użyciem ... 210
Wyrażenia regularne ... 211
Podstawy składni wyrażeń regularnych ... 211
Tworzenie wyrażeń regularnych ... 213
Weryfikacja poprawności adresów e-mail ... 215
Wyrażenia regularne w PHP ... 215
Wyrażenia regularne zgodne z mechanizmami języka Perl ... 218
Funkcje PHP obsługujące PCRE ... 220

Rozdział 8. Sesje oraz ciasteczka ... 225
Sesje ... 226
Uaktywnianie obsługi sesji w PHP ... 226
Zastosowanie sesji PHP ... 227
Uruchamianie sesji ... 228
Rejestrowanie zmiennych sesji ... 228
Tworzenie własnych procedur obsługi sesji ... 230
Ustawianie bazy danych ... 230
Adresy URL ... 235
Bezpieczeństwo ... 235
Ciasteczka ... 236
Bezpieczeństwo ... 237
Zastosowania ciasteczek ... 237
Termin ważności ... 238
Ścieżka ... 239
Domena ... 239
Przykładowa aplikacja wykorzystująca ciasteczka ... 240
setcookie() ... 241
Ustawianie parametru okresu ważności ciasteczka ... 242
Ograniczanie dostępu ... 243
Usuwanie ciasteczka ... 246
Łączenie informacji z ciasteczek ... 246
Problemy z ciasteczkami ... 248
Dodatkowe funkcje obsługi sesji ... 250

Rozdział 9. Obsługa plików ... 253
Pliki ... 253
Otwieranie plików ... 254
Zamykanie plików ... 255
Wypisywanie zawartości plików ... 255
Odczyt zawartości plików ... 256
Zapis do plików ... 257

Nawigowanie po pliku ... 257
Kopiowanie, usuwanie i zmiana nazw plików ... 258
Określanie atrybutów plików ... 259
Katalogi ... 260
Dodawanie i usuwanie katalogów ... 262
Przesyłanie plików z przeglądarki ... 263
Przesyłanie plików za pośrednictwem metody PUT ... 264
Przesyłanie plików z wykorzystaniem metody POST ... 264
Przykładowa aplikacja obsługi systemu plików ... 267
Aplikacja służąca do przechowywania plików online ... 267
Wspólne mechanizmy ... 270
Rejestracja nowego użytkownika ... 272
Logowanie ... 276
Tworzenie katalogów ... 282
Usuwanie katalogu lub pliku ... 282
Wysyłanie plików na serwer ... 283
Przeglądanie plików ... 284
Przeglądanie katalogów ... 285
Wylogowanie ... 286

Rozdział 10. Programowanie klientów FTP ... 287

Uaktywnianie obsługi FTP w PHP ... 288
Rozszerzenia obsługi FTP w PHP ... 288
Tworzenie aplikacji klientów FTP ... 289
Procedury ułatwiające korzystanie z FTP ... 290
Klient FTP oparty na WWW ... 299
Tworzymy klienta FTP ... 308
Przegląd funkcji FTP według zastosowania ... 311
Nawiązywanie i zrywanie połączenia ... 311
Operacje na katalogach ... 312
Obsługa plików ... 312
Alfabetyczny przegląd funkcji FTP ... 313
Podstawowe polecenia FTP oraz odpowiadające im funkcje PHP ... 323

Rozdział 11. Poczta elektroniczna i grupy dyskusyjne ... 327

Jak działa poczta elektroniczna? ... 328
Niezbyt tajni agenci ... 329
SMTP ... 329
Struktura listu elektronicznego ... 331
Nagłówki listu elektronicznego ... 331
Nagłówki wymagane ... 332
Nagłówki opcjonalne ... 334
Wysyłanie listów elektronicznych z wykorzystaniem funkcji mail() ... 335
Wykorzystanie funkcji mail() ... 335
Tworzenie klasy KlasaPocztowa ... 338
Testowanie klasy KlasaPocztowa ... 344
Tworzenie klasy KlasaPocztowaSMTP ... 345
Testujemy klasę KlasaPocztowaSMTP ... 352
List elektroniczny w formacie MIME ... 353
Pola nagłówka listu elektronicznego w formacie MIME ... 354
Tworzenie klasy KlasaPocztowaMIME ... 358
Testowanie klasy KlasaPocztowaMIME ... 362
Tworzenie klasy KlasaPocztowaSMTP_MIME ... 363

Usenet ... 364
Jak działa Usenet? ... 365
Przykładowa sesja NNTP ... 365
Kody odpowiedzi serwera NNTP ... 368
Anatomia artykułu grupy dyskusyjnej ... 370
Tworzenie klasy NNTP ... 371
Testowanie klasy KlasaNNTP ... 377
Łączymy w całość wszystkie elementy ... 378
Dodatkowe źródła informacji ... 385

Rozdział 12. Pobieranie listów elektronicznych i artykułów grup dyskusyjnych ... 387

Protokoły służące do pobierania poczty elektronicznej ... 388
POP ... 388
Przykładowa sesja POP ... 389
IMAP ... 390
Znaczniki ... 391
Formaty skrzynek pocztowych ... 391
Przykładowa sesja IMAP ... 392
Porównanie protokołu POP z IMAP ... 396
Pobieranie poczty elektronicznej za pomocą PHP ... 397
Połączenie z serwerem ... 397
Przykład połączenia ... 399
Tworzenie klasy Webmail ... 400
Atrybuty ... 400
Testowanie klasy Webmail ... 403
Pobieranie zawartości skrzynki pocztowej lub grupy dyskusyjnej ... 403
Pobieranie zawartości skrzynki lub grupy dyskusyjnej w klasie Webmail ... 410
Nowe atrybuty ... 410
Testowanie klasy Webmail ... 413
Pobieranie listów i artykułów ... 414
Odczytywanie listów z wykorzystaniem klasy Webmail ... 416
Nowe atrybuty ... 416
Testujemy klasę Webmail ... 420
Praca ze skrzynkami ... 422
Zarządzanie skrzynkami z wykorzystaniem klasy Webmail ... 425
Nowe atrybuty ... 425
Operacje na listach i artykułach ... 428
Operacje na listach wykonywane z wykorzystaniem klasy Webmail ... 430
Nowe atrybuty ... 430
System obsługi poczty elektronicznej oparty na przeglądarce WWW ... 433
Atrybuty ... 433
Dodatkowe źródła informacji ... 447

Rozdział 13. Sieci i protokół TCP/IP ... 449

Internet Protocol ... 450
Protokoły warstwy transportowej ... 451
TCP — Transmission Control Protocol ... 451
UDP — User Datagram Protocol ... 452
Tłumaczenie nazw domen ... 452
Hierarchiczny system rozproszony ... 453
Wykorzystanie DNS w PHP ... 454
Biblioteka Resolver ... 458

Gniazda ... 463
Gniazda i PHP ... 464
Aplikacja klienta pocztowego ... 469
Network Information Service ... 471
Serwery NIS ... 472
Klienci NIS ... 473
Mapowania NIS ... 473
NIS i PHP ... 475
Simple Network Management Protocol ... 477
Agenci i zarządcy ... 477
Protokół SNMP ... 478
Get ... 478
Get Next ... 479
Set ... 479
Trap ... 479
Organizacja danych SNMP ... 479
Funkcje SNMP w PHP ... 480

Rozdział 14. LDAP ... 485

Katalogi ... 485
LDAP ... 486
LDAP a tradycyjne bazy danych ... 486
Składniki LDAP ... 488
LDAP — charakterystyka ... 488
Globalne usługi katalogowe ... 489
Otwarty standard komunikacyjny ... 489
Rozszerzalność i elastyczność ... 489
Heterogeniczne repozytorium danych ... 489
Bezpieczny protokół z kontrolą dostępu ... 490
Zastosowania LDAP ... 490
Elementy terminologii LDAP ... 492
Modele LDAP ... 493
Model informacyjny ... 493
Model nazw ... 495
Model funkcjonalny ... 496
Zaawansowane cechy LDAP ... 499
Operacje asynchroniczne ... 499
Replikacja ... 499
Odsyłacze ... 500
Bezpieczeństwo ... 500
Właściwości rozszerzone ... 500
Oprogramowanie LDAP ... 501
Instalacja i konfiguracja serwera LDAP ... 502
Plik konfiguracyjny serwera OpenLDAP ... 502
Uruchamianie serwera slapd ... 504
Sprawdzanie instalacji ... 505
Obsługa LDAP w języku PHP ... 505
Interfejs LDAP API języka PHP ... 506
Funkcje połączeniowe i kontrolne ... 506
Funkcje wyszukujące ... 508
Funkcje modyfikujące ... 514
Funkcje obsługi błędów ... 516
Przykładowa aplikacja klienta LDAP w języku PHP ... 516

Rozdział 15. Wprowadzenie do programowania aplikacji wielowarstwowych ... 533
Rozwój aplikacji WWW ... 533
Wielowarstwowość ... 535
Warstwa danych ... 535
Model plikowy ... 536
Model relacyjny ... 537
Model XML ... 538
Model mieszany ... 540
Warstwa logiki aplikacji ... 540
Warstwa prezentacji ... 540
Urządzenia korzystające z sieci WWW ... 541
Architektury projektowania wielowarstwowego ... 541
Architektura oparta na języku HTML ... 542
Warstwa danych ... 543
Warstwa logiczna ... 543
Warstwa prezentacji ... 543
Architektura oparta na języku XML ... 545
Wyodrębnianie warstw ... 546
Programowanie modułowe ... 547
Niezależność warstw logiki i prezentacji ... 547
Niezależność warstw logiki i danych ... 547
Niezależność od bazy danych ... 547
Ankieta — projektowanie aplikacji wielowarstwowej ... 548
Projektowanie modelu danych ... 548
Warstwa danych ... 548
Warstwa logiczna ... 549
Warstwa prezentacji ... 550
Klasyczna architektura wielowarstwowa ... 550
Przypadek 1. Zmiana sposobu wyświetlania wyników głosowania ... 550
Przypadek 2. Zablokowanie możliwości wielokrotnego głosowania użytkownika ... 550
Przypadek 3. Wersja Flash aplikacji ... 551

Rozdział 16. Aplikacja WAP — studium przypadku ... 553
Analiza wymagań ... 553
Interakcja z użytkownikiem ... 555
Dobór oprogramowania ... 555
Alternatywy dla bazy danych zaplecza ... 557
Alternatywy dla warstwy pośredniej ... 557
Projekt schematu bazy danych ... 558
Tabele bazy danych ... 558
Użytkownik bazy danych ... 560
Indeksy ... 561
Kwestie projektowe warstwy pośredniej ... 562
Uwierzytelnianie ... 562
Przechowywanie danych sesji ... 562
Kwestie związane z językiem WML ... 563
Wydajność ... 564
Implementacja ... 564
Kod aplikacji ... 566
Warstwa danych i logiki aplikacji ... 572
Karta powitalna ... 595
Rejestracja nowego użytkownika ... 598
Logowanie ... 599

Karta główna aplikacji ... 601
Przeglądanie zasobów księgarni ... 605
Przeglądanie zasobów sklepu muzycznego ... 607
Wyszukiwanie ... 609
Dodawanie pozycji do koszyka użytkownika ... 612
Przeglądanie zawartości koszyka użytkownika ... 614
Zmiana liczby sztuk poszczególnych pozycji koszyka ... 616
Zatwierdzanie zakupów ... 618
Przeglądanie informacji o koncie użytkownika ... 620
Wylogowanie ... 623

Rozdział 17. PHP i MySQL ... 625

Relacyjne bazy danych ... 626
Indeksy ... 627
Klucze ... 627
Normalizacja ... 629
Strukturalny język zapytań ... 631
Zapytania definicji danych ... 632
CREATE DATABASE ... 632
USE ... 633
CREATE TABLE ... 633
DESCRIBE ... 634
ALTER TABLE ... 636
DROP TABLE ... 637
DROP DATABASE ... 637
Zapytania manipulacji danymi ... 638
INSERT ... 638
REPLACE ... 638
DELETE ... 639
UPDATE ... 639
SELECT ... 640
Połączenia ... 641
Indeksy ... 642
Niepodzielność operacji ... 644
PHP a relacyjne bazy danych ... 645
Interfejs MySQL języka PHP ... 645
Biblioteka sieciowa ... 651
Wyodrębnianie bazy danych ... 658
Warstwa abstrakcji bazy danych ... 660
Konstrukcja klasy BD ... 660
Testowanie klasy BD ... 664

Rozdział 18. PHP i PostgreSQL ... 667

PostgreSQL — podstawy ... 668
Zapytania definicji danych ... 669
CREATE DATABASE ... 669
CREATE TABLE ... 669
ALTER TABLE ... 672
DROP TABLE ... 673
DROP DATABASE ... 673
Zapytania manipulacji danymi ... 673
INSERT ... 673
DELETE ... 674
UPDATE ... 674
SELECT ... 675

Interfejs PostgreSQL języka PHP ... 676
Biblioteka sieciowa ... 683
Wyodrębnianie bazy danych ... 687

Rozdział 19. PHP i ODBC ... 691

ODBC — historia i przeznaczenie ... 692
Architektura ODBC ... 693
Standardy SQL ... 694
Instalacja PHP i ODBC w systemie Windows ... 694
Instalacja ODBC i PHP w systemach z rodziny UNIX ... 695
Moduł serwera Apache ... 695
Interfejs ODBC języka PHP ... 698
Funkcje połączeniowe ... 698
Funkcje manipulujące metadanymi ... 699
Funkcje obsługi transakcji ... 702
Funkcje dostępu do danych i kursory ... 703
Najczęstsze problemy ... 706
Wymagania dla połączeń ODBC ... 708
MS SQL Server ... 708
MS Access ... 710
Nawiązywanie połączenia ... 711
Wyodrębnianie bazy danych ... 713
Unified ODBC ... 713
PEARDB ... 713
ADODB ... 714
Metabase ... 715
Biblioteka sieciowa ... 715

Rozdział 20. Programowanie aplikacji nieserwerowych w języku PHP ... 721

Czym jest GTK? ... 721
Czym jest PHP-GTK? ... 722
Język PHP w wierszu poleceń ... 722
Konfiguracja dla systemu Linux ... 722
Biblioteka libedit ... 722
Instalacja PHP-GTK ... 723
Konfiguracja dla systemu Windows ... 724
Środowisko ... 724
Instalacja PHP-GTK ... 725
Automatyzacja zadań ... 726
Format plików dziennika NCSA CLFF ... 727
Skrypt analizatora pliku dziennika ... 728
cron ... 730
AT ... 731
Harmonogram zadań systemu Windows ... 731
Przyjmowanie parametrów wiersza poleceń ... 731
Skrypty interaktywne ... 732
Programowanie aplikacji z użyciem PHP-GTK ... 734
PHP-GTK — podstawy ... 734
Przykład — program Hello World ... 737
Interfejs graficzny aplikacji biblioteki sieciowej ... 739
Dodatkowe źródła informacji ... 746

Rozdział 21. PHP i XML 747
XML — przegląd 748
Rodzina standardów XML 750
XML a bazy danych 751
SML 752
Konwersja dokumentu XML na format SML 752
XML i PHP 754
Weryfikacja obsługi XML-a 754
Porównanie interfejsów XML 755
SAX a DOM 756
PRAX a DOM i SAX 756
Model SAX 756
Obsługa SAX w języku PHP 758
SAX — kod przykładowy 758
Model DOM 764
Obsługa DOM w języku PHP 764
DOM — kod przykładowy 766
Model RAX 775
Obsługa PRAX w języku PHP 776
XSL i XSLT 780
Sablotron 781
Instalacja i weryfikacja konfiguracji XSL 781
Instalacja w systemie UNIX 781
Instalacja w systemie Windows 781
XSL — kod przykładowy 782

Rozdział 22. Internacjonalizacja aplikacji 787
Internacjonalizacja — pojęcia 787
Internacjonalizacja 788
Lokalizacja 788
Obsługa języka ojczystego 789
Motywy internacjonalizacji aplikacji 789
W czym tkwi problem? 790
Ciągi tekstowe 790
Ciągi statyczne 791
Ciągi dynamiczne 791
Przechowywanie ciągów 792
Wyodrębnienie danych tekstowych z programu 793
GNU Gettext 794
Gettext — informacje podstawowe 794
xgettext i inne narzędzia pomocnicze 795
Aktualizacja tłumaczeń 798
Wady biblioteki Gettext 798
Obiektowe rozszerzenie systemu tłumaczenia 799
Zalety podejścia obiektowego 799
Korzystanie z obiektów i przełączanie języków 800
Konwersja istniejących programów 800
Program nieprzetłumaczony 801
Tłumaczenie programu 801
Obiekty przystosowane do wielojęzyczności przekładu 804
Integracja klasy Wyjscie z aplikacją 806
Dalsze doskonalenie skryptu 808
Wyrażenia regularne 808

Wielkość liter ... 810
Lokalne formaty daty i czasu ... 810
Pozyskiwanie dodatkowych informacji lokalizacyjnych — funkcja localeconv() ... 813
Sortowanie ... 816
Własna funkcja porównująca ... 817
Kodowanie znaków ... 819
Zapisywanie locali ... 819
Przeglądarka a język ... 819
Reagowanie na konfigurację przeglądarki ... 821
Ciągi poszerzone ... 825
Moduł obsługi ciągów poszerzonych w języku PHP ... 826
Moduł mod_mime serwera Apache ... 826
Przykład lokalizacji rzeczywistej aplikacji — PHP Weather ... 826

Rozdział 23. Bezpieczeństwo aplikacji PHP ... 831

Czym jest bezpieczeństwo? ... 832
Zabezpieczanie serwera ... 832
Zbrojenie serwera ... 833
Monitorowanie systemu ... 833
Monitorowanie powiadomień o nowych lukach ... 834
Najpowszechniejsze rodzaje zagrożeń ... 834
Zabezpieczanie serwera Apache ... 836
Dyrektywa User ... 836
Dyrektywa Directory ... 837
Zbrojenie serwera Apache ... 838
Zabezpieczanie PHP ... 839
Bezpieczeństwo instalacji PHP w trybie CGI ... 839
Konfiguracja PHP ... 840
Tryb bezpieczny ... 843
Zabezpieczanie serwera MySQL ... 844
MySQL i użytkownik root ... 844
Sprzątanie ... 845
Zarządzanie użytkownikami baz danych MySQL ... 846
Kryptografia ... 847
Szyfrowanie jednokierunkowe ... 847
Szyfrowanie symetryczne ... 850
Szyfrowanie asymetryczne ... 852
Bezpieczeństwo sieci komputerowej ... 852
Moduł mod_ssl serwera Apache ... 853
Instalacja modułu mod_ssl w systemie Linux ... 853
Instalacja modułu mod_ssl w systemie Windows ... 854
Konfiguracja modułu mod_ssl ... 854
Kiedy należy korzystać z połączeń SSL? ... 855
Bezpieczne programowanie ... 855
Zagrożenia związane z dyrektywą register_global ... 856
Kontrola danych wprowadzanych przez użytkownika ... 858
Zagrożenia płynące z nieuprawnionego wykonania kodu HTML ... 859
Pułapki dyrektywy include ... 859
Kilka porad ... 860
Dodatkowe źródła informacji ... 861
Zabezpieczanie serwerów linuksowych ... 861
Bezpieczne powłoki (SSH) ... 861

Tripwire ... 861
Zabezpieczanie Apache ... 862
Zabezpieczanie PHP ... 862
Zabezpieczanie MySQL ... 862
Kryptografia ... 862
mod_ssl ... 863
Bezpieczne programowanie ... 863
Strony WWW poświęcone bezpieczeństwu ... 863
Pozostałe źródła ... 863

Rozdział 24. Optymalizacja aplikacji PHP ... 865

Właściwy język ... 865
Wyniki testów ... 866
Optymalizacja kodu PHP ... 867
Profilowanie kodu ... 867
Profilowanie skryptów PHP ... 867
Klasyfikacja wąskich gardeł ... 871
Techniki optymalizacji ... 871
Optymalizacja kodu ... 872
Analiza pętli ... 872
Wykorzystanie szybszych funkcji ... 873
Wybór właściwego sposobu przekazywania danych wyjściowych ... 873
Wybór właściwego sposobu pobierania danych wejściowych ... 873
Minimalizacja liczby wywołań funkcji echo() ... 874
Wykorzystanie optymalizatora Zend Optimizer ... 874
Buforowanie i kompresja danych wyjściowych ... 874
Przykład skryptu buforującego wyjście ... 875
Funkcje obsługujące buforowanie wyjścia ... 875
Buforowanie kaskadowe ... 877
Kompresja wyjścia skryptu PHP ... 877
Optymalizacja bazy danych ... 878
Analiza zapytań ... 878
Szacowanie efektywności zapytania ... 878
Optymalizacja tabel ... 883
Optymalizacja modelu danych ... 883
Stosowanie indeksów ... 884
Optymalizacja zapytań SELECT ... 885
Optymalizacja zapytań INSERT ... 885
Optymalizacja zapytań UPDATE ... 886
Optymalizacja zapytań DELETE ... 886
Optymalizacja połączeń ... 886
Optymalizacja — wskazówki dodatkowe ... 887
Buforowanie wyników obliczeń ... 887
Czym jest buforowanie? ... 888
Waga buforowania ... 888
Zalety buforowania ... 888
Wady buforowania ... 888
Ogólna metodologia buforowania ... 889
Wybór metody składowania buforowanych danych ... 890
Konwencje nazewnicze ... 892
Kryteria poprawności ... 892
Opróżnianie bufora ... 893
Jakie dane powinny być składowane w buforach wyników? ... 893
Optymalizacja interpretera PHP ... 895

Rozdział 25. Biblioteki rozszerzeń języka PHP ... 897
Biblioteka PDF ... 898
Instalacja ... 898
Korzystanie z biblioteki PDFlib ... 899
Macromedia Flash ... 903
Ming i LibSWF ... 903
Korzystanie z biblioteki Ming ... 904
WAP i WML ... 912
A gdzie tu biblioteka? ... 913
Korzystanie z biblioteki HAWHAW ... 914
Tworzenie dynamicznych rysunków ... 918
Instalacja biblioteki GD ... 918
Korzystanie z biblioteki GD ... 919
Licznik odwiedzin wykonany za pomocą GD ... 921
Kod licznika odwiedzin ... 921

Rozdział 26. System uprawnień użytkowników ... 925
Definicja wymagań ... 925
Wymagania aplikacji ... 926
Projektowanie aplikacji ... 926
Projektowanie schematu bazy danych ... 927
Tabela Uzytkownik ... 927
Tabela Uprawnienie ... 927
Tabela UzytkownikUprawnienie ... 927
Projektowanie warstwy pośredniczącej ... 927
Dostęp do bazy danych ... 928
Klasa Uprawnienie ... 928
Klasa Uzytkownik ... 928
Warstwa logiki aplikacji ... 929
Projektowanie warstwy prezentacji ... 930
Kodowanie aplikacji ... 931
Kod obsługi bazy danych ... 931
Klasa Uprawnienie ... 932
Klasa Uzytkownik ... 934
Testowanie klas ... 938
uprawnienia.php ... 939
uzytkownikuprawnienia.php ... 944
Wykorzystanie systemu uprawnień użytkowników ... 950
Kilka propozycji rozszerzenia systemu ... 951

Skorowidz ... 953

5

Programowanie obiektowe w PHP4

Programowanie zorientowane obiektowo (*Object Oriented Programming*) istnieje już od kilku lat. Jego historia wiąże się z powstaniem języków Smalltalk oraz C++. Koncepcja programowania obiektowego zaowocowała następnie powstaniem nowych języków, takich jak Java czy Python. Gdy w grę wchodzi tworzenie skomplikowanych programów, takich jak edytor tekstu czy też gra komputerowa, programowanie obiektowe nie jest zaledwie jedną z opcji dostępnych programiście. Jest to obecnie standardowy sposób tworzenia skomplikowanych programów w sposób ułatwiający ich rozwijanie i zwiększający poziom ich skalowalności, stosowany zarówno w oprogramowaniu komercyjnym, jak i bezpłatnym.

Usprawnienie mechanizmów obiektowych w PHP4 spowodowało znaczne ożywienie w społeczności programistów PHP. Zaczęli oni wykorzystywać zalety programowania obiektowego. W tym rozdziale przyjrzymy się technikom programowania obiektowego od podstaw, pokazując, jaką rolę odgrywają one w PHP, a także wskazując sposoby ich wykorzystania w celu tworzenia eleganckich rozwiązań w dziedzinie aplikacji WWW. Przyjrzymy się kilku sposobom analizy oraz pożytecznym praktykom programistycznym pozwalającym na zwiększenie możliwości powtórnego wykorzystania kodu, a także uproszczenia procesu dalszego rozwoju programów. Zanim jednak zagalopujemy się za daleko, zobaczmy, w jaki sposób powstała koncepcja programowania obiektowego, a także czym różni się ona od tradycyjnych technik programowania proceduralnego. Jeśli jesteś doświadczonym programistą stosującym techniki obiektowe i chcesz zapoznać się z zagadnieniami obiektowymi specyficznymi dla PHP, możesz od razu przejść do podrozdziału „Klasy".

Programowanie zorientowane obiektowo

Na początku prześledźmy różnice pomiędzy programowaniem obiektowym a klasycznym programowaniem proceduralnym. Przed powstaniem idei programowania obiektowego programy stawały się coraz bardziej rozbudowane i skomplikowane. Systemy te dla dalszego rozwoju wymagały pracy wielu architektów i inżynierów, a proces ten pochłaniał coraz więcej czasu i pieniędzy. Często zachodzi potrzeba dopasowania oprogramowania do zmian w strategii biznesowej. W takim przypadku zmiana istniejących funkcji programu lub dodanie nowych wymagały tygodni, a często miesięcy pracy i okazywało się, że napisanie programu od nowa trwałoby znacznie krócej.

Aplikacje rozrastały się, a usuwanie błędów w nich stawało się poważnym problemem. Wysiłki w celu utrzymania istniejących możliwości systemu zajmowały więcej czasu niż dodawanie nowych funkcji. Kod stawał się coraz bardziej chaotyczny, ponieważ rosła liczba pracowników zaangażowanych w projekt. Błędy projektowe popełnione na początku (co często ma miejsce w przypadku projektów programistycznych prowadzonych w językach takich jak Fortran czy C) były często przyczyną nawarstwiania się problemów. W wielkich firmach, inwestujących ogromne pieniądze w systemy informatyczne niezbędne do prowadzenia biznesu, uświadomiono sobie ogromną potrzebę usprawnienia procesu projektowania i tworzenia oprogramowania.

Wtedy właśnie naukowcy z wielu dziedzin, takich jak informatyka, filozofia czy biologia, stworzyli podstawy nowego sposobu tworzenia oprogramowania, który został ostatecznie nazwany „programowaniem zorientowanym obiektowo". W procesie tym najbardziej zasłużyli się Alan Kay, twórca języka Smalltalk, oraz Grady Booch, autor nowoczesnych zasad analizy i programowania obiektowego.

Celem tych prac było zażegnanie kryzysu w przemyśle programistycznym przez stworzenie łatwego w użyciu zestawu narzędzi programistycznych. Wspomniani pionierzy techniki obiektowej odkryli, że rozwiązanie problemów programistycznych jest możliwe przez zastosowanie kilku nowych zasad tworzenia oprogramowania, wymagających od programisty nieco więcej wysiłku na początku pracy. Programowanie obiektowe zmusza programistę do spojrzenia na problemy i sposoby ich rozwiązania z nieco innej perspektywy. Kosztem zastosowania technik obiektowych jest niewielkie zmniejszenie wydajności oprogramowania, ale jego dalsza rozbudowa staje się niezwykle łatwa.

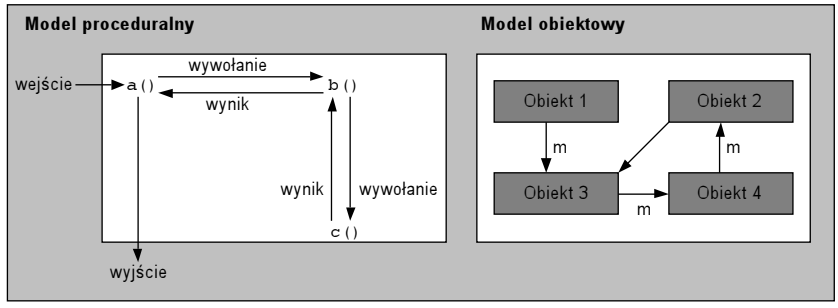
Oczywista jest konieczność kompromisu pomiędzy wydajnością i łatwością rozbudowy. W przypadku niektórych systemów komputerowych, w których krytycznym czynnikiem jest działanie w czasie rzeczywistym lub też które wykorzystują operacje wymagające bardzo dużej wydajności, zastosowanie rozwiązań obiektowych nie wchodzi w grę — właśnie z powodu wymagań dotyczących czasu wykonania. Jednak wszędzie tam, gdzie zastosowanie programowania obiektowego jest uzasadnione, programista może tworzyć programy w sposób niezwykle elastyczny i o wiele bardziej intuicyjny niż do tej pory. Programowanie zorientowane obiektowo wspomaga tworzenie oprogramowania w sposób ułatwiający ponowne wykorzystanie kodu, rozbudowę, wykrywanie błędów i zwiększający czytelność programów.

Wraz z rozwojem technik obiektowych powstawały nowe metodologie prowadzenia projektów informatycznych, takie jak osławione programowanie ekstremalne (*eXtreme Programming*) czy też ujednolicony proces (*Unified Process*). Łatwiej jest obecnie planować fazy projektów, gospodarować zasobami, wtórnie wykorzystywać istniejący kod, a także testować i analizować kod w większych projektach. Faktem jest, że powstanie technik programowania obiektowego miało ogromny wpływ na nasz dzisiejszy, oparty na elektronice świat. Technologie takie jak Java wykorzystują techniki obiektowe do granic możliwości, udostępniając pojedyncze, zorientowane obiektowo rozwiązanie przeznaczone dla różnych platform, począwszy od urządzeń elektronicznych codziennego użytku, poprzez oprogramowanie dla komputerów klasy PC, na potężnych aplikacjach internetowych skończywszy.

Programowanie proceduralne a programowanie obiektowe

A zatem co powoduje, że programowanie obiektowe tak różni się od programowania proceduralnego? Gdy tworzymy oprogramowanie przy użyciu procedur (funkcji), tworzymy programy **zorientowane na kod**. Programy wywołują funkcje, które wywołują funkcje i tak dalej. Dane są przesyłane jako argument wywołania funkcji, funkcja dokonuje modyfikacji danych i zwraca odpowiednie wyniki. Programowanie zorientowane obiektowo to zupełnie inna metodologia. Tutaj programowanie jest **zorientowane na dane**. Obiekty, które wewnętrznie reprezentują dane, zawierają dodatkowo mechanizmy funkcji, zwanych w tym przypadku **metodami**.

Metoda jest usługą (w swej implementacji niezwykle podobną do funkcji), którą obiekt udostępnia swoim klientom (innym obiektom). Gdy obiekt wysyła żądanie usługi do drugiego obiektu, odbywa się to na zasadzie przesłania komunikatu i uzyskania odpowiedzi. Oto porównanie obydwu metod:



Wprowadzenie danych powoduje wywołanie funkcji `a()`, która wywołuje funkcję `b()`. Funkcja `b()` wywołuje `c()`, która zwraca swój wynik do `b()`, która z kolei zwraca swój wynik funkcji `a()`. Funkcja `a()` ostatecznie generuje wynik programu. Funkcja `a()` w typowym programie w `c()` nosiłaby nazwę `main()`. W modelu obiektowym obiekt wysyła żądania usług do innych obiektów, co widać na przykładzie. Tutaj `Obiekt 1` wysyła żądanie do `Obiekt 3`. Z kolei `Obiekt 3` wysyła żądanie do `Obiekt 4` i tak dalej, aż do momentu, gdy `Obiekt 1` uzyska odpowiedź od `Obiekt 3` zawierającą wynik końcowy.

Ten mechanizm opiera się na udostępnianiu usług obiektom przez inne obiekty celem przekazywania informacji niezbędnej do ich pracy. Oznacza to możliwość podejmowania decyzji na podstawie informacji uzyskanych z innych obiektów. Przekazywanie komunikatów stanowi samo w sobie przebieg programu. Dane i metody, a także sam kod implementujący obiekty, są zgromadzone w jednym centralnym miejscu.

Różnica pomiędzy tymi dwoma sposobami polega na tym, że obiekty zawierają całe dane oraz mechanizmy działania, które powinny być ze sobą związane, natomiast w rozwiązaniu proceduralnym dane i mechanizmy działania istnieją niezależnie od siebie. Cecha ta pozwala na łatwą analizę kodu wykorzystującego technikę obiektową i zwiększa modularność projektu.

Nie oznacza to jednak, że programy wykorzystujące rozwiązanie proceduralne nie dają się łatwo rozwijać. Wymaga to jednak znacznie więcej uwagi i organizacji ze strony programisty, aby zapewnić odpowiednie umiejscowienie wszystkich elementów. Najlepszą rzeczą

w programowaniu obiektowym jest to, że tworzone obiekty mają sens w projekcie, opierają się na jasnych założeniach, dzięki czemu wszystko wydaje się być dobrze zorganizowane. W bardziej skomplikowanych projektach można wykorzystać wyspecjalizowane rozwiązania, które dodatkowo usprawnią projekt naszego systemu, zapewniając dodatkowe korzyści.

Znaczenie programowania obiektowego

Programista musi sobie uświadomić, że programowanie obiektowe jest czymś więcej niż techniką programistyczną. Nie jest to język ani platforma systemowa. PHP, C++ czy Java są językami obiektowymi, ponieważ wykorzystują to samo rozwiązanie, choć każdy z nich realizuje to na swój unikalny sposób. Nie zmienia to faktu, że programowanie w C++ różni się dość znacznie od programowania w PHP, ponieważ języki te różnią się składnią oraz działaniem konstrukcji językowych.

Jednak ponieważ języki zorientowane obiektowo są skonstruowane wg podobnych zasad, podstawy programowania obiektowego we wszystkich tych językach są podobne. Dlatego należy najpierw nauczyć się podstaw samych koncepcji obiektowych, aby następnie zagłębić się w szczegóły implementacji tych technik w wybranym języku programowania. Przy bliższej obserwacji okazuje się, że PHP obsługuje jedynie podzbiór cech oczekiwanych od obiektowego języka programowania. Ograniczenia PHP w tej dziedzinie zostaną omówione w dalszej części tego rozdziału, a pod koniec rozdziału przedstawimy ich podsumowanie.

Zstępująca metoda tworzenia oprogramowania

Od czasów powstania nauki ludzie wykazują skłonność do dzielenia na kategorie, definiowania i formalizowania wszystkiego, co istnieje na świecie. Programowanie nie jest tu wyjątkiem, ponieważ informatyka wyrasta z silnych korzeni matematyki i logiki. Piękno programowania obiektowego polega na tym, że pozwala ono nie tylko umieścić dane i kod na swoim miejscu, ale również umożliwia podzielenie na kategorie i zdefiniowanie elementów programów w sposób, w jaki postrzegamy elementy naszego świata. Technika ta umożliwia kompleksowe opracowanie projektu, zanim zagłębimy się w szczegóły. Pozwala to na łatwe oszacowanie czasu, zagrożeń i różnych zasobów, które mają związek z naszym projektem.

Gdy tworzymy program, możemy podzielić go na wyspecjalizowane części lub moduły. Możemy na przykład wyodrębnić różne warstwy prezentacji, warstwę dostępu do bazy danych, mechanizmy wyszukiwania czy też komponenty odpowiedzialne za bezpieczeństwo. Gdy tworzymy nasze moduły jako wyodrębnione elementy, zyskujemy pewność, że zmiany dokonane w jednym z obiektów nie będą miały wpływu na drugi. Dodatkowo będziemy mogli ponownie wykorzystać obiekty w innych aplikacjach. Możemy również z łatwością rozbić nasze moduły na podmoduły, a w końcu na pojedyncze klasy, które stanowią najmniejszy komponent w programowaniu zorientowanym obiektowo. Przyjrzyjmy się teraz temu najmniejszemu komponentowi w programie zorientowanym obiektowo, czyli klasie.

Klasy

Klasa jest definicją reprezentacji określonego typu danych. Klasy służą jako sposób modelowania różnych typów danych w naszym systemie. Gdy chcemy utworzyć obiekt, najpierw użyjemy słowa kluczowego `class`, aby zdefiniować go, zanim użyjemy go w naszym skrypcie w PHP. Różnica pomiędzy klasą a obiektem polega na tym, że klasa definiuje obiekty używane w programie. Zanim dowiemy się, w jaki sposób tworzyć klasy, musimy zacząć myśleć o klasach jak o reprezentacji pojedynczej idei. W trakcie projektowania klasy trzeba starać się, aby realizowała konkretny cel w sposób jak najbardziej kompletny, nie wykraczając jednak poza przeznaczenie przypisane tej idei.

Klasa w PHP składa się z trzech głównych elementów: atrybutów (reprezentujących dane), metod oraz konstruktorów. **Atrybut** stanowi porcję danych przechowywanych przez obiekt. Obiekty mogą zawierać dowolną liczbę atrybutów — na przykład jeśli chcielibyśmy przedstawić samochód w postaci klasy, kierownica i skrzynia biegów byłyby atrybutami klasy `Samochod`. **Metody** określają usługi, które obiekt udostępnia swoim klientom w celu manipulowania atrybutami. Przykładowo klasa `Samochod` może udostępniać metody umożliwiające wykonanie manewru skrętu pojazdem, wykorzystując wewnętrzny atrybut reprezentujący kierownicę.

Konstruktor jest specjalną metodą inicjującą obiekt do stanu gotowości. W PHP w każdej klasie może być zdefiniowany tylko jeden konstruktor. W klasie `Samochod` sensowne wydaje się dodanie atrybutów `nadwozie`, `silnik`, `kola`, `skrzynia_biegow`, `siedzenia` i tym podobne. Gdy klient zechce użyć metod obiektu, konstruktor zapewnia, że każda z metod będzie działać na prawidłowych atrybutach i zwróci oczekiwany wynik, na przykład aby można było włączyć radio w samochodzie, musi być ono zamontowane. W tym przypadku konstruktor jest odpowiedzialny za zamontowanie radia przed jego użyciem.

Oprócz realizowania funkcji inicjowania obiektu do stanu gotowości, dodatkową różnicą między konstruktorami a metodami jest to, że konstruktory nie zwracają jawnie żadnych wartości. Każdy konstruktor zwraca nowo zainicjowany obiekt, gotowy do użycia w programie. Dlatego użycie instrukcji `return` w definicji konstruktora jest nielegalne. W dalszej części rozdziału zajmiemy się szerzej problematyką wykorzystania obiektów w programach.

Często spotykanym problemem jest konieczność opracowania prawidłowego projektu obiektów i konstruktorów. Jeśli konstrukcja klasy zmusza programistę do „ręcznego" inicjowania atrybutów przed ich użyciem lub też zachowania określonej kolejności inicjowania atrybutów, prowadzi to do powstania zagmatwanego i nieczytelnego kodu. Programowanie zorientowane obiektowo pozwala uniknąć takich sytuacji. Jeśli klasa zostanie zaprojektowana w sposób nie wykorzystujący konstruktora do inicjacji kluczowych atrybutów, projekt taki uznajemy za nieprawidłowy. Uważaj, aby nie wpaść w tę pułapkę.

> Dobrze zaprojektowana klasa może nam zaoszczędzić wielu problemów w trakcie programowania, usuwania błędów i rozwijania projektu.

Przyjrzyjmy się ogólnej składni definicji klas w PHP, demonstrującej wykorzystanie trzech typów komponentów:

```
class NazwaKlasy [extends NazwaKlasyNadrzednej]
{
    var $atrybut1;
    var $atrybut2;
    ...
    var $AtrybutN;

    // Konstruktor
    function NazwaKlasy() {
    }

    function metoda1() {
    }

    function metoda2() {
    }
...
    function metodaN() {
    }
}
```

Jak widać, klasa jest po prostu zestawem definicji atrybutów (zmiennych) i metod (funkcji). Atrybuty mogą być zmiennymi typów prostych, takich jak liczby czy napisy, lub bardziej złożonych, jak tablice czy inne obiekty. Ponieważ PHP nie wymaga definicji typów, wystarczy po prostu wymienić nazwy atrybutów na początku definicji klasy, tak jak w powyższym przykładzie.

PHP umożliwia tworzenie nowych atrybutów w metodach w trakcie wykonania programu i będą one działać prawidłowo. Jest to jednak uważane za zły nawyk programistyczny. Inni programiści powinni podczas analizy kodu poznać wszystkie atrybuty klasy na podstawie definicji klasy bez konieczności zagłębiania się w szczegóły implementacji metod.

Metody stanowią wszelkie usługi udostępniane przez tę klasę jej klientom. Klientami mogą być inne programy, obiekty, skrypty itd.

Utwórzmy kod dla klasy `Samochod`. W tym przykładzie rozpoczynamy definicję klasy, używając słowa kluczowego `class`. Za dobrą praktykę programistyczną uważa się rozpoczynanie wszystkich nazw klas z wielkich liter, w celu odróżnienia ich od nazw zmiennych i funkcji.

Programiści przestrzegają tej zasady od lat. Dzięki temu łatwo jest odróżnić konstruktor od innych metod w klasie. Innym dobrym zwyczajem jest nazywanie plików tak samo, jak klasy, na przykład *Samochod.php*. Pojedynczy plik powinien zawierać definicję tylko jednej klasy. Jeśli masz wiele klas odwołujących się wzajemnie do siebie — tworzą kolekcję klas, powinieneś umieścić je w katalogu głównym swojej aplikacji. W przypadku dużych projektów taki zwyczaj staje się wymogiem.

> Gdy projekt się rozrośnie, konieczne stanie się wykorzystanie struktury drzewiastej w celu przechowywania klas wykorzystywanych w aplikacji. Będziesz zmuszony do wykorzystania dyrektyw include_once() lub require_once(), aby włączyć pliki definicji tych klas do swoich skryptów.

```
<?php
//Samochod.php
class Samochod
{
```

W nieprawdopodobnie prostym modelu samochodu klasa składa się z silnika i stacyjki rozrusznika służącej do uruchomienia pojazdu. Prawdziwy samochód będzie składał się z nadwozia, drzwi, pedałów gazu, sprzęgła, hamulca, kierownicy, skrzyni biegów i wielu, wielu innych elementów. W naszym przypadku chodzi jednak jedynie o demonstrację:

```
    var $silnik;
    var $stacyjka;
```

Nasz samochód również posiada konstruktor inicjujący silnik i stacyjkę. Gdybyśmy nie zainicjowali tych elementów, każde użycie metod `start()` lub `stop()` zakończyłoby się błędem. Jak wspomnieliśmy wcześniej, zadaniem konstruktora jest zainicjowanie wszystkich elementów obiektu w celu zapewnienia, że wszystkie usługi będą mogły zostać wykorzystane wtedy, gdy będą potrzebne.

Aby odwołać się do atrybutu, należy użyć konstrukcji `$this->` przed nazwą atrybutu. Jest to różnica w porównaniu z Javą i C++, gdzie takie odwołanie do obiektu klasy jest opcjonalne. PHP wymaga tego zapisu, ponieważ język ten nie jest dobrze wyposażony w obsługę zakresów widzialności zmiennych. W PHP istnieją trzy poziomy **przestrzeni nazw**, w których przechowywane są zmienne (przestrzeń nazw jest, ogólnie rzecz biorąc, kolekcją nazw zmiennych).

Na najniższym poziomie występuje przestrzeń nazw używana w celu przechowywania lokalnych zmiennych funkcji i metod. Każda zmienna definiowana na tym poziomie jest dodawana do lokalnej przestrzeni nazw. Kolejna przestrzeń nazw zawiera atrybuty obiektów. Przestrzeń nazw najwyższego poziomu jest używana do przechowywania zmiennych globalnych. Konstrukcja `$this->` wskazuje, że odwołujemy się do zmiennej z przestrzeni nazw obiektu (środkowa warstwa). Jeśli zapomnisz zastosować zapis `$this->`, utworzysz zupełnie nową zmienną w lokalnej przestrzeni nazw. Ponieważ będzie ona odwoływać się do zupełnie innej wartości, mogą powstać trudne do wychwycenia błędy logiczne.

> Koniecznie uaktywnij opcję raportowania błędów, którą omówimy w następnym rozdziale. Dodatkowo dodaj kilka asercji, aby zabezpieczyć się przed tym błędem, tak powszechnym przy pracy z klasami.

Metoda `start()` uruchomi samochód użytkownikowi posiadającemu kluczyk, jeśli kluczyk ten jest właściwy.

```
// Konstructor
function Samochod()
{
    $this->stacyjka = new ZwyklaStacyjka();
    $this->silnik = new Silnik();
}

function start($kluczyk) {
    if ($kluczyk->pasuje($this->stacyjka)) {
```

```
        $this->silnik->start();
        return true;
    }
    return false;
}
```

Metoda `stop()` sprawdza, czy silnik działa, i jeśli tak, zatrzymuje samochód. Zauważmy, że sprawdzenie stanu uruchomienia samochodu mogłoby zostać przeniesione do metody `stop()` obiektu silnika, co zwolniłoby nas z konieczności obsługi tego przypadku. Projektując klasy, często napotkasz takie rozterki dotyczące umiejscowienia logiki obiektów. Podejmowanie właściwych decyzji projektowych jest podstawą dobrego i skutecznego tworzenia aplikacji.

```
function stop()
{
    if ($this->silnik->pracuje()) {
        $this->silnik->stop();
    }
}
// inne metody, na przykład obsługa skręcania, ruszania itp.
}
?>
```

Teraz przedstawimy, w jaki sposób można stosować obiekty w programach.

Obiekty

Obiekt w programie jest **egzemplarzem** klasy. Obiekty nazywamy egzemplarzami, ponieważ można utworzyć wiele obiektów (czyli egzemplarzy) tej samej klasy — tak jak na drodze może być wiele samochodów tego samego typu. Aby utworzyć dwa nowe samochody, musimy wykonać następujące dwa wiersze kodu:

```
<?php
$samochod1 = new Samochod();
$samochod2 = new Samochod();
```

W celu utworzenia nowego egzemplarza klasy (to znaczy utworzenia obiektu) używamy słowa kluczowego `new`. Odwołania do nowo utworzonych obiektów zostają umieszczone w zmiennych `$samochod1` oraz `$samochod2`. Mamy więc dwa obiekty samochodów, których możemy użyć. Gdybyśmy chcieli utworzyć dziesięć samochodów, moglibyśmy użyć tablicy obiektów:

```
$samochody = array();
for ($i = 0; $i < 10; $i++) {
    $samochody[$i] = new Samochod();
}
```

Gdy zechcemy uruchomić samochód, wywołujemy jego metodę `start()` w następujący sposób:

```
$samochodUruchomiony = $samochod->start($mojKluczyk);
if ($samochodUruchomiony) echo("Samochód został uruchomiony.");
```

A gdy zechcemy zatrzymać samochód, piszemy:

```
$samochod->stop();
?>
```

Zauważ, że interfejs naszego obiektu jest łatwy w użyciu. Nie musimy martwić się o to, jak został zaimplementowany. Jako programista musisz tylko znać serwisy udostępniane przez obiekt. Ten program mógłby z powodzeniem sterować procesem uruchamiania i zatrzymywania silnika prawdziwego samochodu. Szczegóły kryjące się za metodami, a także atrybutami mogą pozostać zupełnie nieznane. Koncepcja tworzenia łatwych w użyciu obiektów doprowadza nas do tematu następnego podrozdziału, czyli hermetyzacji. Na razie zajmiemy się jednak tworzeniem egzemplarzy obiektów z wykorzystaniem metod fabrycznych.

Metody fabryczne

Czasem wygodniej jest poprosić obiekt o utworzenie innego obiektu, zamiast samemu wywoływać konstruktor. Klasy udostępniające takie usługi nazywamy **klasami fabrycznymi,** a metody używane do tworzenia obiektów nazywamy **metodami fabrycznymi**. Określenie *fabryczny* wywodzi się ze skojarzenia z fabryką — instytucją wytwarzającą gotowe wyroby. Na przykład fabrykę silników General Motors, produkującą silniki do samochodów, można porównać z obiektem fabrycznym tworzącym obiekty określonego typu. Nie zagłębiając się w szczegóły zaawansowanych technik obiektowych, zademonstrujemy teraz możliwe zastosowanie obiektów fabrycznych w tworzeniu aplikacji WWW. Można je zastosować do:

- zdefiniowania obiektu tworzącego elementy formularzy (takie jak pola tekstowe, grupy opcji, przyciski formularzy itd.) celem umieszczenia go w formularzu HTML. Klasa ta będzie realizować zadania analogiczne do zadań klasy dostępnej w ramach biblioteki *eXtremePHP* (dostępnej na stronie *http://www.extremephp.org/*);
- zdefiniowania obiektu fabrycznego wstawiającego do tabeli bazy danych nowe wiersze i zwracającego odpowiedni obiekt dostępu do danych zawartych w tym konkretnym wierszu.

Przyjrzyjmy się teraz sposobowi definiowania klas fabrycznych na przykładzie tworzenia obiektów `PoleTekstowe` oraz `PrzyciskPotwierdzenia` z poziomu klasy `FabrykaElementowFormularza` (zapożyczonej z biblioteki *eXtremePHP*).

W przykładzie wykorzystujemy dwa pliki zawierające klasy. Zakładamy, że zostały one utworzone wcześniej. Plik *PoleTekstowe.php* zawiera kod definiujący klasę `PoleTekstowe`, natomiast *Przycisk.php* zawiera kod klasy `Przycisk`. Konstruktory tych klas wymagają podania nazwy i wartości tworzonego elementu:

```
<?php
include_once("./PoleTekstowe.php");
include_once("./Przycisk.php");
```

Dobrym zwyczajem programistycznym jest dodawanie przyrostka „Factory" na końcu nazwy klasy. Zachowanie tej konwencji, stosowanej powszechnie w środowisku programistów wykorzystujących techniki obiektowe, pozwoli innym programistom na pierwszy rzut oka zidentyfikować zadanie klasy:

```
// ElementFormularzaFactory.php
class ElementFormularzaFactory
{
```

Naszą pierwszą metodą fabryczną jest `utworzPoleTekstowe()`. Metoda ta po prostu tworzy pole tekstowe, przekazując parametry `$nazwa` oraz `$wartosc` przekazane przez klienta.

```
    function utworzPoleTekstowe($nazwa, $wartosc)
    {
        return new PoleTekstowe($nazwa, $wartosc);
    }
```

Metoda `utworzPrzycisk()` zostanie zdefiniowana w podobny sposób. Warto używać podanej przez nas (lub podobnej) konwencji nazywania metod fabrycznych (przedrostek `utworz`), aby jednoznacznie zasygnalizować innym programistom, że przeznaczeniem metody jest utworzenie nowego obiektu. Taka praktyka ujednolici terminologię w tworzonych aplikacjach i polepszy przejrzystość kodu.

```
    function utworzPrzycisk($nazwa, $wartosc)
    {
        return new Przycisk($nazwa, $wartosc);
    }
}
```

Teraz zamiast tworzyć egzemplarze klas `PoleTekstowe` oraz `Przycisk` za pomocą operatora `new`, możemy skorzystać z obiektu klasy `ElementFormularzaFactory`, aby dokonał tego za nas:

```
$fabrykaElementowFormularza = new ElementFormularzaFactory();
$poleTekstoweImie =
    $fabrykaElementowFormularza->utworzPoleTekstowe('imie', 'Jan');
$poleTekstoweNazwisko =
    $fabrykaElementowFormularza->utworzPoleTekstowe('nazwisko', 'Kowalski');
$przyciskZatwierdz =
    $fabrykaElementowFormularza->utworzPrzycisk('zatwierdz', 'Zatwierdź imię
    ➥i nazwisko');
?>
```

Utworzyliśmy egzemplarz klasy `ElementFormularzaFactory`, a następnie utworzyliśmy trzy nowe obiekty z użyciem metod obiektu fabrycznego. Pierwsze dwa wywołania metody `utworzPoleTekstowe()` powodują utworzenie pól tekstowych do wpisania imienia i nazwiska. Wywołanie metody `utworzPrzycisk()` powoduje utworzenie przycisku z napisem `Zatwierdź imię i nazwisko`. W tym momencie nasza aplikacja może wykorzystać utworzone obiekty do dowolnych celów. Nie chodziło nam jednak o znaczenie aplikacji, lecz o zastosowanie obiektów fabrycznych w aplikacjach internetowych.

Klasy fabryczne nie ograniczają się jedynie do udostępniania metod tworzących obiekty. Można swobodnie dodawać metody, które wydają się zasadne w modelu fabrycznym, takie jak metody wyszukujące obiekty w naszej „fabryce" czy metody usuwające, które umożliwiają pozbywanie się obiektów z zasobów fabryki. Decyzje dotyczące takich metod są ściśle uzależnione od projektu i zależą wyłącznie od projektanta aplikacji. Teraz przyjrzyjmy się zasadom hermetyzacji i ukrywania informacji.

Hermetyzacja (ang.Encapsulation)

Gdy zażywasz środek przeciw bólowi głowy, zwykle nie wiesz, z czego się składa. Jedyne, co Cię interesuje, to jego działanie polegające na usunięciu bólu głowy. W podobny sposób programiści wykorzystują obiekty. Gdy stosowaliśmy obiekt klasy `Samochod`, nie wiedzieliśmy niczego o jego skrzyni biegów, układzie wydechowym czy też silniku. Chcieliśmy tylko przekręcić kluczyk i uruchomić samochód. Umożliwienie takiego wykorzystania obiektu powinno stać się głównym celem jego projektanta.

Wszelkie skomplikowane dane i logikę obiektu należy ukryć wewnątrz obiektu, udostępniając jedynie ważne usługi służące do posługiwania się obiektem. W ten sposób dokonujemy hermetyzacji skomplikowanych danych i szczegółów implementacyjnych wewnątrz obiektu. Jeśli dokonamy tego właściwie, osiągniemy dodatkowy zysk w postaci ukrycia informacji.

Jak wspomnieliśmy wcześniej, nieświadomość istnienia oraz rodzaju atrybutów danych przechowywanych w strukturze klasy może stanowić korzyść dla wykorzystujących ją programistów.

> Mimo że modyfikacja atrybutów obiektu jest poprawną operacją w PHP, jest to uznane za zły nawyk programistyczny.

Zademonstrujemy kilka możliwych sytuacji, do których może doprowadzić modyfikacja atrybutów obiektu bez korzystania z pośrednictwa interfejsu obiektu. W naszym przykładzie zakładamy, że istnieje metoda ustalająca prędkość samochodu o nazwie `Predkosc ($predkosc)`, która kontroluje przekroczenie prędkości 200 km/h, a także sprawdza, czy prędkość nie jest mniejsza od zera. Załóżmy również, że konstruktor nie zainicjował silnika i stacyjki:

```
$mojKluczyk = new Kluczyk('Kluczyk od Porszaka');
$samochod = new Samochod();
$samochod->silnik = new Silnik();

$samochod->predkosc = 400;
$samochod->start($mojKluczyk);

$samochod->silnik = 0;
$samochod->stop();
```

Powyższy kod zawiera wiele błędów, które w konsekwencji spowodują błąd interpretera lub, co gorsza, spowodują, że program będzie działać nieprawidłowo. W pierwszych trzech wierszach nie utworzyliśmy kluczyka pasującego do naszego samochodu, ponieważ nie skorzystaliśmy z naszego konstruktora.

Kluczyk nie jest potrzebny przed uruchomieniem samochodu, więc nie pojawią się żadne błędy. Następnie przyjrzyjmy się fragmentowi, w którym tworzymy obiekt klasy `Silnik`. Co stałoby się, gdybyśmy zamiast tego fragmentu wpisali `$samochod->Silnik = new Silnik()` (zwróć uwagę na wielką literę w błędnej nazwie atrybutu `Silnik`)? Samochód również nie działałby prawidłowo, ponieważ silnik nie zostałby zainicjowany. Oczywiście, błędy tego typu można łatwo odnaleźć i usunąć, ale nie powinny się one w ogóle pojawić. Następnie usiłujemy uruchomić samochód:

```
$samochod->predkosc = 400; // aby zadziałała kontrola, powinniśmy użyć
                           // $samochod->ustalPredkosc(400)
$samochod->start($mojKluczyk);
```

Gdy samochód zostanie uruchomiony, wystartuje z prędkością 400 km/h. Może to spowodować wypadek i śmierć wielu użytkowników ruchu drogowego. Oczywiście, nie chcemy dopuścić do wystąpienia takiej sytuacji.

Skąd samochód wie, jaki rodzaj kluczyka jest wymagany do jego uruchomienia? W tym celu porówna nasz poprawnie skonstruowany kluczyk ze zmienną, która nie istnieje (co w wyniku daje wartość `0`) i w konsekwencji spowoduje, że samochód nie uruchomi silnika. Taki błąd nie zostanie wykryty przez interpreter, ponieważ porównujemy zmienną `$kluczyk`, nie kontrolując przynależności do klasy. Dziwną sytuacją byłoby zakupienie nowiutkiego samochodu od dealera tylko w celu sprawdzenia, że kluczyk nie współpracuje ze stacyjką. Na końcu sprawdźmy, co może się stać, gdy przypiszemy wartość `0` atrybutowi `silnika`:

```
$samochod->silnik = 0;
$samochod->stop();
```

Gdy wywoływana jest metoda `stop()`, zostaje wywołany błąd wykonania, ponieważ atrybut `silnik` klasy `Samochod` nie jest obiektem klasy `Silnik`, gdyż wymusiliśmy zmianę wartości atrybutu `$silnik` na `0`. Jak widać, przypisywanie wartości atrybutom bez pośrednictwa metod klasy może spowodować wiele problemów. Gdy nad projektem pracuje wielu programistów, należy przewidzieć sytuację, w której Twój kod będzie czytany, a nawet używany przez innych programistów.

Jaka nauka płynie z powyższego przykładu? Używanie atrybutów obiektu bezpośrednio może mieć następujące konsekwencje:

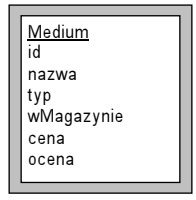
- brak kontroli wykorzystania atrybutów w przewidywalny sposób;
- naruszenie integralności atrybutów obiektu (lub stanu obiektu) w jeden z następujących sposobów:
  - pogwałcenie zasad określających poprawne wartości danych;
  - pominięcie inicjacji atrybutów;
- tworzenie zbyt skomplikowanych interfejsów;
- zmuszenie programisty do zapamiętania większej liczby informacji na temat powiązań atrybutów z metodami;
- w przypadku ponownego wykorzystania obiektu może pojawić się potrzeba modyfikacji atrybutów. Może to doprowadzić do ponownych błędów podczas tworzenia nowego projektu, a właśnie tego chcielibyśmy uniknąć.

Dobrą zasadą jest takie zaprojektowanie klasy, aby udostępniała usługi realizujące wszelkie operacje na obiekcie. Należy unikać bezpośredniej modyfikacji atrybutów obiektu z zewnątrz i zawsze stosować techniki hermetyzacji danych, aby wykorzystać zalety ukrywania informacji. Niektóre języki oferują techniki blokowania dostępu do atrybutów obiektów z zewnątrz przez deklarowanie ich jako prywatne (*private*) lub chronione (*protected*). W obecnej implementacji PHP nie obsługuje żadnej z tych technik, ale na pewno pomocne okaże się przestrzeganie ustalonych zasad programistycznych.

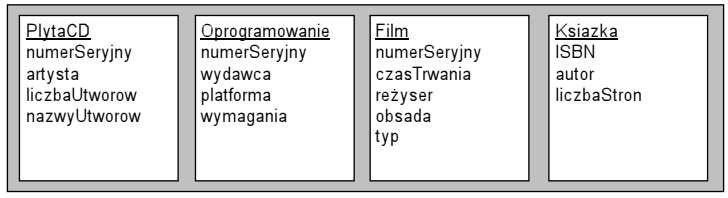
Dziedziczenie

Zapoznaliśmy się już z podstawowymi elementami programowania zorientowanego obiektowo i z niektórymi jego dobrymi praktykami. Nadszedł czas, aby zapoznać się z mechanizmami udostępnianymi przez programowanie obiektowe, pozwalającymi na podejmowanie przejrzystych strategii rozwiązywania skomplikowanych problemów.

Załóżmy, że naszym celem jest uporządkowanie i zarządzanie sklepem internetowym typu *amazon.com*. Chcielibyśmy sprzedawać płyty CD, oprogramowanie, kasety VHS, płyty DVD oraz książki. Korzystając z tradycyjnych rozwiązań opartych na funkcjach, możemy utworzyć tradycyjną strukturę do przechowywania informacji o tych mediach, na przykład:



Oczywiście, istnieje mnóstwo różnic pomiędzy książkami, filmami, płytami CD i oprogramowaniem, więc zechcemy utworzyć dodatkowe struktury przechowujące dane dla określonych typów medium:



Aby napisać program wypisujący elementy tablicy zawierającej nasze media, zastosujemy następujący kod:

```
<?php
// tablica zawierająca 2 elementy
$elementy = array();
$ksiazki = array();
$plytyCd = array();

$element->id = 1;
$element->typ = "ksiazka";
$element->nazwa = "Programowanie w PHP4 dla profesjonalistów";
$element->wMagazynie = 33;
$element->cena = 49.95;
$element->ocena = 5;
Selementy[] = $element;
$ksiazka->isbn = 1246534343443;
$ksiazka->autor = "Ken Egervari, i inni";
$ksiazka->liczbaStron = 500;
$ksiazki[$element->id] = $ksiazka;
```

```
$element->id = 2;
$element->typ = "cd";
$element->nazwa = "This Way";
$element->wMagazynie = 120;
$element->cena = 16.95;
$element->ocena = 4;
$elementy[] = $element;

$cd->numerSeryjny = 323254354;
$cd->artysta = "Jewel";
$cd->liczbaUtworow = 13;
$plytyCd[$element->id] = $cd;

// wypisanie elementów z tablicy
foreach ($elementy as $element) {
    echo("Nazwa: " . $element->nazwa . "<br>");
    echo("Ilość sztuk w magazynie: " . $element->wMagazynie . "<br>");
    echo("Cena: " . $element->cena . "<br>");
    echo("Ocena: " . $element->ocena . "<br>");

    switch ($element->typ) {
    case 'cd':
        echo("Numer seryjny: " . $ plytyCd[$element->id]->serialNo . "<br>");
        echo("Wykonawca: " . $plytyCd[$element->id]->wykonawca . "<br>");
        echo("Liczba utworów: " . $ plytyCd[$element->id]->liczbaUtworow . "<br>");
        break;

     case 'software':
         // wypisujemy informacje specyficzne dla oprogramowania
         break;

   case 'film':
         // wypisujemy informacje specyficzne dla filmów
         break;

   case 'ksiazka':
         // wypisujemy informacje specyficzne dla książek
         break;
      }
}
```

A jeśli zechcemy dodać jeszcze jeden typ medium? Musielibyśmy wrócić do tego kodu, dodać jeszcze jedną instrukcję `case` i prawdopodobnie uaktualnić w podobny sposób kod w wielu innych miejscach naszej aplikacji. Programowanie zorientowane obiektowo udostępnia nam technikę zwaną **dziedziczenie**, która umożliwia umieszczenie szczegółów specyficznych dla poszczególnych typów obiektów w jednym miejscu, a także pozwala na zgrupowanie cech wspólnych dla wszystkich typów obiektów. Wykorzystując tę technikę, możemy zupełnie wykluczyć instrukcję `switch` w przypadkach podobnych do rozpatrywanego przez nas.

W naszym przykładzie sklepu multimedialnego możemy wyodrębnić podobieństwa pomiędzy rodzajami mediów w jednej klasie obiektów. Klasa taka nosi nazwę **klasy macierzystej**, **klasy podstawowej** lub klasy **nadrzędnej**. W ten sposób utworzymy najbardziej abstrakcyjną implementację (atrybuty i metody), która odnosi się do każdego elementu, jaki chcemy umieścić w naszym sklepie. Poniżej przedstawiamy fikcyjną implementację klasy `Medium`:

```
<?php
define("MIN_OCENA", 0);
define("MAX_OCENA", 5);
// Media.php
class Medium {
    var $id;
    var $nazwa;
    var $wMagazynie;
    var $cena;
    var $ocena;
    function Medium($id, $nazwa, $wMagazynie, $cena, Socena) {
        if ($wMagazynie < 0) $wMagazynie = 0;
        if ($cena < 0) $cena =0;
        if ($ocena < MIN_OCENA) $ocena = MIN_OCENA;
        if ($ocena > MAX_OCENA) $ocena = MAX_OCENA;
        $this->id = $id;
        $this->nazwa = Snazwa;
        $this->wMagazynie = $wMagazynie;
        $this->cena = $cena;
        $this->ocena = $ocena;
    }
    function kup() {
        $this->wMagazynie--;
    }
    function pokaz() {
        echo("nzwa: " . $this->nazwa . "<br>");
echo("Liczba sztuk w magazynie: " . $this->wMagazynie . "<br>");
        echo("Cena: " . $this->cena . "<br>");
        echo("Ocena: " . $this->ocena . "<br>");
    }
    // więcej metod
}
?>
```

Teraz, gdy mamy zdefiniowaną klasę macierzystą, możemy zastosować słowo kluczowe `extends`, aby odziedziczyć atrybuty i metody po tej klasie i zdefiniować klasę potomną klasy `Medium` wyspecjalizowaną pod kątem konkretnego typu mediów, na przykład książek lub filmów. Klasa wyspecjalizowana, powstała wskutek dziedziczenia po klasie macierzystej, nazywana jest **klasą potomną** lub **podklasą**. Przedstawiamy klasę `Ksiazka`, która jest klasą potomną klasy `Medium`. Pozostałe klasy można zdefiniować w podobny sposób.

> Dobrą praktyką programistyczną jest tworzenie podklas na podstawie elementu, który w przypadku podejścia proceduralnego wymuszałby wiele rozgałęzień kodu. W naszym przykładzie jest to atrybut `$typ`, który zmuszał nas do tworzenia rozbudowanych instrukcji `case`. Wyeliminowanie tego atrybutu i utworzenie klasy na podstawie informacji, które przechowywał, pozwoli nam znacznie ograniczyć komplikacje logiki w naszej aplikacji.

```
<?php
// Ksiazka.php
class Ksiazka extends Medium
{
var $isbn;
var $autor;
var $liczbaStron;
```

```
function Ksiazka($id, $nazwa, $wMagazynie, $cena, $ocena,
$isbn, $autor, $liczbaStron) {
// Ważne jest, aby najpierw wywoływać konstruktor klasy rodzimej,
// a dopiero potem przypisywać wartości własnych atrybutów
$this->Medium($id, $nazwa, $wMagazynie, $cena, $ocena);
 $this->isbn = $isbn;
 $this->autor = $autor;
 $this->liczbaStron = $liczbaStron;
}
function pokaz() {
    Medium::pokaz();
    echo("ISBN: " . $this->isbn. "<br>");
    echo("Autor: " . $this->autor. "<br>");
    echo("Liczba stron: " . $this->liczbaStron . "<br>");
}
// metody
}
?>
```

Gdy klasa `Ksiazka` dziedziczy po klasie `Medium`, automatycznie zawiera atrybuty i metody klasy macierzystej. Aby utworzyć nowy obiekt klasy `Ksiazka`, w jej konstruktorze stosujemy konstruktor klasy macierzystej `Medium()`, a następnie inicjujemy własne atrybuty. Taki projekt uznaje się za elegancki, ponieważ oszczędza nam inicjacji wielu atrybutów, które są już inicjowane w konstruktorze klasy macierzystej. Dotyczy to w szczególności logiki kontrolującej poprawność danych przekazanych w argumentach konstruktora. Na przykład, wartość atrybutu `$wMagazynie` nie powinna być mniejsza od `zera`, a atrybut `$cena` nie powinien być liczbą ujemną. Umieszczając tę logikę w metodach klasy `Medium`, możemy mieć pewność, że wszystkie klasy potomne będą również miały zapewnioną tę samą integralność danych.

Zwróćmy uwagę na metodę `wypisz()` w klasie `Ksiazka`. Udostępniamy tutaj nową implementację metody wypisującej informacje o obiekcie klasy `Ksiazka`. W tej nowej metodzie wypisujemy atrybuty klasy `Medium` (używając operatora wywołania metody klasy, opisanego w dalszej części rozdziału), jak również atrybuty specyficzne dla tej klasy. Nowa metoda `wypisz()` przesłania metodę `wypisz()` klasy macierzystej `Medium`.

Ponieważ nasze klasy potomne posiadają wspólny interfejs udostępniony przez klasę `Medium`, możemy używać ich w ten sam sposób, co powoduje, że są łatwiejsze w użyciu, a także łatwiej jest je dalej rozwijać. Poniżej przedstawiamy kod wypisujący informacje o dwóch obiektach: pierwszego klasy `Ksiazka` i drugiego klasy `Cd`.

```
$ksiazka = new Ksiazka(0, "Programowanie w PHP", 23, 59.99, 4, "1124-4333-4443",
➥'Ken Egervari', 1024);
$ksiazka->wypisz();
$cd = new Cd(1, "Positive Edge", 1911, 16.99, 5, "Ken Egervari", 10,
➥$tablicaNazwUtworow);
$cd->wypisz();
```

Zwróćmy uwagę, że wszystkie nasze obiekty klasy `Medium` zachowują się identycznie. Udostępniają metodę `wypisz()`, która nie pobiera argumentów i wypisuje informacje o odpowiednich elementach, ponieważ obiekty „wiedzą", jakiego są typu. Pozwala to na opracowanie wspólnego interfejsu dla różnych typów obiektów. W jaki sposób może to pomóc nam wypisywać informacje o różnych elementach w naszym sklepie? Zademonstrujemy to w podrozdziale zatytułowanym „Polimorfizm" w dalszej części rozdziału. Teraz jednak przyjrzyjmy się jeszcze kilku interesującym zagadnieniom dotyczącym dziedziczenia.

Operator wywołania metody klasy

Przyjrzyjmy się nowemu operatorowi oznaczanemu jako podwójny dwukropek (::), który pozwala na wywołanie metody określonej klasy bez potrzeby tworzenia nowego obiektu. W ten sposób dla wywoływanej metody klasy nie będzie dostępny żaden atrybut ani konstruktor. Składnia wywołania wygląda następująco:

```
NazwaKlasy::nazwaFunkcji();
```

Powyższa instrukcja po prostu wywoła metodę klasy `NazwaKlasy` o nazwie `nazwaFunkcji()`. Jeśli metoda o podanej nazwie nie istnieje w definicji klasy, interpreter PHP zasygnalizuje błąd.

Wróćmy do przykładu klasy `Ksiazka`. W metodzie `wypisz()` wykorzystaliśmy wywołanie:

```
Medium::wypisz()
```

Wywołanie to wywołuje metodę `wypisz()` klasy `Medium`. Ponieważ nasza klasa `Ksiazka` (jak również `Cd`, `Film`, `Oprogramowanie`) dziedziczy po klasie `Medium`, w efekcie wykorzystujemy metodę `wypisz()` klasy macierzystej. To spowoduje wypisanie informacji o obiekcie klasy `Medium`, po czym możemy uzupełnić te informacje o specyficzne dla klasy `Ksiazka`, wykorzystując instrukcję `echo`.

Jeśli `NazwaKlasy` nie jest nazwą klasy macierzystej obiektu, w którym nastąpiło wywołanie, metoda zostanie użyta w sposób statyczny, to znaczy nie będą dostępne atrybuty klasy `NazwaKlasy`. To może być użyteczne w celu zgrupowania podobnych funkcji w obrębie klasy. Możemy zdefiniować klasę `Math` definiującą metody `floor()`, `ceil()`, `min()` oraz `max()`. Jest to przykład banalny, niemniej jednak ilustruje możliwość grupowania funkcji, które powinny być zgrupowane. Zamiast wywołania:

```
$c = floor(1.56);
```

możemy zastosować klasę `Math` w następujący sposób:

```
$c = Math::floor(1.56) ;
```

Zaletą jest zgrupowanie podobnych funkcji w ramach klasy `Math`, co pozwala na łatwiejszą modyfikację kodu w późniejszym okresie. Ponadto kod staje się bardziej „zorientowany obiektowo", udostępniając wszelkie zalety programowania obiektowego.

Wadą takiego rozwiązania jest to, że wyrażenia będą dłuższe, ponieważ za każdym razem musimy dodawać nazwę klasy przed nazwą wywoływanej metody.

Ponowne wykorzystanie kodu

Dziedziczenie udostępnia dobry sposób ponownego wykorzystania kodu, lecz głównym zadaniem tej techniki jest specjalizacja klasy w wyniku dodania nowych możliwości do już istniejących w ramach klasy macierzystej. Ponowne wykorzystanie kodu nie jest zadaniem, do realizacji którego zostało wymyślone dziedziczenie w programowaniu obiektowym, choć jest ono jedną z zalet specjalizacji klas. Zadaniem dziedziczenia jest udostępnienie

możliwości wykorzystania klas potomnych w podobny sposób. Nie martw się jednak, ponieważ przyjrzymy się innym sposobom ponownego wykorzystania kodu w podrozdziale zatytułowanym „Delegacja".

> Dobrą praktyką programistyczną jest niewykorzystywanie dziedziczenia wyłącznie w celu uzyskania korzyści z ponownego wykorzystania kodu.

Wszystkie klasy potomne (takie jak `Cd`, `Ksiazka` czy `Film`) będą posiadały tę samą liczbę danych i funkcji, co klasa macierzysta, lub będą posiadały dodatkowe atrybuty i metody, niedostępne w klasie macierzystej. Innymi słowy, klasy potomne są zwykle „lepiej wyposażone" w funkcje od ich klasy macierzystej.

Polimorfizm

Polimorfizm jest kolejną cechą programowania obiektowego, która pozwala na traktowanie obiektów w jednolity sposób. Możemy wykorzystać możliwości istniejących klas, a także dowolnych ich klas potomnych zdefiniowanych w przyszłości, pozostawiając interpreterowi wszelkie szczegóły dotyczące różnic pomiędzy klasami. Polimorfizm powoduje, że bardzo łatwo jest dodawać nowe klasy do systemu bez wprowadzania błędów w istniejącym kodzie.

Wróćmy do przykładu ze sklepem z artykułami multimedialnymi. Gdy użytkownik systemu zażąda listy nowych artykułów, nie powinno mieć znaczenia, czy to książka, film czy płyta CD. W tym miejscu wkracza właśnie polimorfizm. Technika ta pozwala traktować obiekty reprezentujące wymienione media w jednakowy sposób. Nie musimy kłopotać się sprawdzaniem typów zmiennych przy użyciu instrukcji `if` lub `switch`. Zadanie to pozostawiamy interpreterowi PHP.

Polimorfizm jest techniką łatwą do opanowania, gdy rozumiemy już zasady dziedziczenia, ponieważ jedynym sposobem uzyskania zjawiska polimorfizmu jest dziedziczenie. Dziedziczenie pozwala na tworzenie abstrakcyjnej klasy macierzystej, a następnie implementowanie metod tej klasy w klasach potomnych. W naszym przykładzie wszystkie klasy potomne posiadają własne metody służące prezentacji informacji na ich temat. Nasza klasa macierzysta zawiera metodę `wypisz()`, co gwarantuje nam, że wszelkie klasy potomne będą posiadać tę samą metodę. Oto przykład:

```
<?php
$elementy = array(new Ksiazka(...), new Cd(...), new Ksiazka(...), new Cd(...));
foreach ($elementy as $element) {
    $element->wypisz();
    echo("<br><br>") ;
}
```

Powyższy kod wypisze kolejno informacje na temat każdego obiektu medium, niezależnie od tego, czy to jest CD, książka, czy też film lub też program komputerowy. Pomimo istnienia różnic w klasach potomnych klasy `Medium`, możemy traktować je podobnie, ponieważ wszystkie posiadają metodę `wypisz()`. Interpreter PHP rozpozna sytuację i zdecyduje, co należy zrobić.

Przedstawiony kod demonstruje eleganckie rozwiązanie problemu wypisania listy artykułów w sklepie. Jest ono o wiele prostsze od rozbudowanego rozwiązania opierającego się na technice przedstawionego wcześniej programowania proceduralnego. Załóżmy, że musimy dodać klasę o nazwie `graNaKonsole`. Dodamy nową klasę potomną klasy `Medium` i utworzymy kilka obiektów nowej klasy w tablicy `$elementy`, a nasza metoda wypisania elementów nie będzie wymagać żadnych zmian:

```
$elementy[] = new graNaKonsole(...);
foreach ($elementy as $element) {
    $element->wypisz();
    echo("<br><br>");
}
?>
```

Powyższy kod wypisze istniejącą listę, a także nowo utworzony obiekt klasy `graNaKonsole`. Zwróć uwagę na to, że nie musieliśmy wprowadzić jakichkolwiek zmian w instrukcji `foreach`. Polimorfizm umożliwia tworzenie takiego właśnie kodu, łatwego w rozbudowie i zrozumieniu. Dziedziczenie samo w sobie posiada niewiele zalet. Używamy dziedziczenia, aby skorzystać z zalet polimorfizmu, który pozwala nam tworzyć kod przejrzysty i łatwy w rozbudowie. Dziedziczenie nie jest tylko sposobem na ponowne wykorzystanie kodu, jak zauważyliśmy wcześniej, jest przede wszystkim sposobem na udostępnienie polimorfizmu w kodzie aplikacji. Gdy zapoznamy się z różnymi układami projektowymi, uzmysłowimy sobie, jak wiele problemów programistycznych można rozwiązać dzięki wykorzystaniu dziedziczenia i polimorfizmu.

Metody abstrakcyjne

Gdy stosujemy dziedziczenie, często klasa macierzysta definiuje metody nie zawierające kodu, ponieważ zdefiniowanie mechanizmów w postaci wspólnej dla wszystkich klas potomnych nie jest możliwe. Technikę taką, zwaną **metodami abstrakcyjnymi**, wykorzystujemy, aby zaznaczyć fakt, że metoda nie zawiera kodu i programista implementujący klasy potomne musi zaimplementować mechanizmy tych metod. Jeśli metoda nie zostaje przesłonięta (co zostało omówione w podrozdziale „Dziedziczenie"), nadal nie będzie realizować żadnych zadań.

Jeśli nie zdefiniujemy metody abstrakcyjnej, a klasy potomne nie przesłonią jej, zostanie wywołany błąd wykonania zgłaszający brak metody w obiekcie. Dlatego ważne jest definiowanie pustej metody w sytuacji, gdy nie chcemy lub nie możemy zdefiniować żadnego mechanizmu działania metody.

> W terminologii obiektowej zdefiniowanie metody jako abstrakcyjnej z reguły wymusza na programiście przesłonięcie jej w klasie potomnej. PHP nie obsługuje takiego wymuszenia z powodu ograniczeń w zaimplementowanym modelu obiektowym. Osoby implementujące klasy potomne powinny zapoznać się z dokumentacją klas macierzystych, aby określić, które z metod powinny zostać przesłonięte.

Pomimo braku słów kluczowych w PHP, które definiowałyby metody abstrakcyjne, przyjęła się notacja używana przez programistów dla wskazania metody abstrakcyjnej. Aby wskazać osobie implementującej klasy potomne, że metoda jest abstrakcyjna i powinna zostać przesłonięta, definiujemy ją jako pustą. W naszym przykładzie metoda abstrakcyjna jest zaznaczona pogrubioną czcionką:

```
<?php
// Pracownik.php
class Pracownik
{
    var $imie;
    var $nazwisko;
    function Pracownik($imie, $nazwisko) {
        $this->imie = $imie;
        $this->nazwisko = $nazwisko;
    }
    function pobierzImie() {
        return $this->imie;
    }
    function pobierzNazwisko() {
        return $this->nazwisko;
    }
    // metoda abstrakcyjna
    function pobierzMiesieczneZarobki() {}
}
?>
```

W klasie `Pracownik` metodę `pobierzMiesieczneZarobki()` zdefiniowaliśmy jako pustą, aby zaznaczyć, że jest to metoda abstrakcyjna. Przyczyną zdefiniowania tej metody jako abstrakcyjnej jest brak możliwości określenia sposobu wyliczania zarobków każdego pracownika w jednolity sposób. Gdy w firmie istnieją różne stanowiska, takie jak dyrektorzy, handlowcy, inżynierowie i pracownicy produkcyjni, płaca każdego z pracowników jest naliczana w inny sposób. Przeanalizuj, w jaki sposób określamy płacę dyrektora:

```
<?php
require_once("Pracownik.php");

// Dyrektor.php
class Dyrektor extends Pracownik
{
    var $pensja;

    function Pracownik($imie, $nazwisko, $pensja) {
        Pracownik::Pracownik($imie, $nazwisko);
        $this->ustalPensje($pensja);
    }

    function ustalPensje($pensja) {
        if ($pensja < 0) $pensja = 0;
        $this->pensja = $pensja;
    }

    function pobierzMiesieczneZarobki()
    {
        return $this->pensja;
    }
}
?>
```

Metodę abstrakcyjną `pobierzMiesieczneZarobki()` klasy `Pracownik` przesłaniamy metodą obsługującą działanie specyficzne dla dyrektorów. W naszym przykładzie po prostu zwracamy wartość atrybutu `$pensja`.

Handlowiec może otrzymywać miesięczną pensję, ale może też otrzymywać dodatek obliczany na podstawie wielkości zysku, który przyniósł firmie w danym miesiącu. Poniżej przedstawiamy implementację klasy `Handlowiec`, która realizuje powyższe założenia:

```
<?php
require_once("Dyrektor.php");

define("DOMYSLNA_PREMIA", .15);

// Handlowiec.php
class Handlowiec extends Dyrektor {
    var $pensja;
    var $premia; // wartość od 0 do 1
    var $wartoscSprzedazy; // liczba zmiennoprzecinkowa

    function $Handlowiec($imie, $nazwisko, $pensja,
                         $premia, $wartoscSprzedazy) {
        Dyrektor::Dyrektor($imie, $nazwisko, $pensja);
        $this->ustalPremie($premia) ;
        $this->ustalWartoscSprzedazy($wartoscSprzedazy);
    }

    function ustalPremie($premia) {
        if ($premia < 0 || $premia > 1) $premia = DOMYSLNA_PREMIA;
        $this->premia = $premia;
    }

    function ustalWartoscSprzedazy($wartoscSprzedazy) {
        if ($wartoscSprzedazy < 0) $wartoscSprzedazy = 0;
        $this->wartoscSprzedazy = $WartoscSprzedazy;
    }

    function pobierzMiesieczneZarobki() {
        return Dyrektor::pobierzMiesieczneZarobki() +
        ($this->premia * $this->wartoscSprzedazy);
    }
}
?>
```

W naszej klasie `Handlowiec` wykorzystujemy mechanizmy klasy `Dyrektor` i dodajemy dwa atrybuty, `$premia` oraz `$wartoscSprzedazy`. Atrybut `$premia` określa procent od wartości sprzedaży, który służy wyliczeniu premii doliczanej do pensji w celu obliczenia wypłaty, natomiast `$wartoscSprzedazy` stanowi wartość sprzedaży dokonanej przez naszego handlowca w miesiącu, za który obliczamy zarobki. Sposób obliczania miesięcznych zarobków różni się więc od sposobu przyjętego dla dyrektora, dlatego inna jest też implementacja metody `pobierzMiesieczneZarobki()`.

Podobnie możemy postąpić z płacami inżynierów i pracowników produkcyjnych. Poniżej przedstawiamy przykład metody `pobierzMiesieczneZarobki()` dla pracownika rozliczanego na podstawie godzin pracy:

```
function pobierzMiesieczneZarobki() {
    return $this->liczbaprzepracowanychGodzin * $this->stawkaZaGodzine;
}
```

Jak widać w powyższych przykładach, klasa `Pracownik` nie może definiować logiki obliczania miesięcznych zarobków w klasie macierzystej, dlatego rozwiązanie tego problemu pozostawiono klasom potomnym.

Adekwatność i powiązania

Omówiliśmy już sposób przechowywania danych wewnątrz obiektów, a także wykorzystaliśmy więcej niż jedną klasę w celu rozwiązania problemu. W tym miejscu napotykamy pojęcia **adekwatności** oraz **powiązań** klas.

Adekwatność jest stopniem dostosowania atrybutów i metod do potrzeb obiektu. Czy metody i atrybuty klasy są ściśle powiązane, powodując, że obiekt jest w dużym stopniu dostosowany do swojego przeznaczenia, czy też obiekty tej klasy są zmuszone do realizacji setek różnych zadań?

Niektórzy programiści nie zdają sobie sprawy z potencjału drzemiącego w programu obiektowym, choć być może są skuteczni w tworzeniu oprogramowania wykorzystującego obiekty. Utworzenie modułu i „opakowanie" go w klasę nie wystarczy, by rozwiązanie takie było naprawdę zorientowane obiektowo. Obiekty tworzone w ten sposób w rzeczywistości realizują zadania całego programu w ramach jednej klasy. Przyjrzyjmy się przykładowi, w którym pominięto implementację metod, aby zademonstrować podejście do programowania obiektowego charakteryzujące się niskim poziomem adekwatności, polegające na implementacji całości w postaci pojedynczej „klasy boskiej". W tym przykładzie zademonstrujemy mechanizm formularzy, który sprawdza poprawność wprowadzanych danych, wypisuje kod formularza, a także generuje kod JavaScript do obsługi formularza:

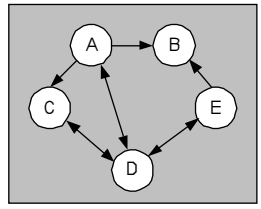
```
class mechanizmFormularzy {
    var $formularze;
    var $elementyFormularzy;
    var $styleFormularzy;
    var $formularz;
    function utworzFormularz() {}
    function dodajElementFormularza($formularz, $nazwa, $wartosc, $atrybuty) {}
    function sprawdzPoprawnoscFormularza($formularz) (}
    function sprawdzPoprawnoscElementu($elementFormularza) {}
    function generujKodJavaScript($elementyFormularza) {}
    function pobierzKodJavaScriptElementuFormularza($elementFormularza) {}
    function wypiszFormularz($formularz) {}
    function wypiszElementFormularza($elementFormularza) {}
    function ustalStylFormularza($formularz) {}
}
```

Powyższy kod to klasa zawierająca struktury i metody przetwarzające wszystko, co ma związek z formularzami. W wielu spośród tych metod będzie występować instrukcja `switch` z dziesięcioma lub więcej elementami, w zależności od poziomu komplikacji obsługiwanych formularzy, wliczając w to kilka specjalnych kombinacji, takich jak elementy formularza obsługujące przesyłanie plików, wprowadzanie dat i tak dalej. Aby obsłużyć przypisywanie stylu do elementów formularzy, również trzeba będzie zastosować instrukcję `switch`. Jaka jest więc różnica między tym sposobem a rozwiązaniem proceduralnym, z dużą ilością zmiennych globalnych zamiast atrybutów i użyciem zwykłych funkcji w miejsce metod? Absolutnie żadna.

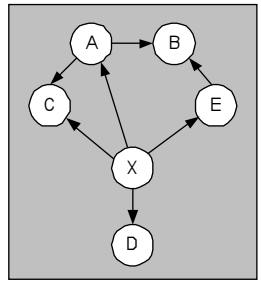
Jak widać w powyższym przykładzie, zastosowanie nieadekwatnych klas prowadzi do powstania „klas boskich" i stanowi błędne podejście do rozwiązania problemu. Przedstawiony przykład nie wykorzystuje zalet programowania obiektowego. Gdy projektujesz klasę, staraj się ją zminimalizować tak, jakby to była funkcja. Najlepszym podejściem jest specjalizacja funkcji w sposób, który zapewnia jak najdoskonalszą realizację pojedynczego zadania. Dokładnie ta sama zasada dotyczy obiektów. Gdy wystąpi konieczność wprowadzenia zmian w kodzie, będziesz wiedział dokładnie, gdzie go szukać. To znacznie upraszcza proces usuwania błędu, oszczędzając nam konieczności przeglądania tysięcy wierszy kodu w ramach pojedynczego pliku. Dlatego należy poświęcić sporo uwagi projektowi klas i odpowiedniemu oddzieleniu logiki od danych, ponieważ tworzenie wysoce adekwatnych obiektów jest w dalszej perspektywie bardzo korzystne.

Przenieśmy naszą dyskusję na zagadnienia związane z powiązaniami.

Pojęcie powiązań dotyczy liczby wzajemnych relacji pomiędzy dwoma lub większą liczbą obiektów. Gdy mamy do czynienia z dwoma obiektami, które są świadome wzajemnego istnienia, mówimy o **silnych powiązaniach**. Można to porównać do błędnego zdefiniowania kanałów komunikacyjnych celem wymiany informacji pomiędzy kilkoma osobami. W tym przypadku każdy ma powiązania z prawie każdym:



Powyższy diagram przedstawia pięć obiektów, z których każdy ma powiązania z przynajmniej jednym z pozostałych. Strzałki określają kierunek komunikacji (jedno bądź dwukierunkową). Obiekt *D* zawiera kod, który zakłada istnienie obiektów *A*, *C* oraz *E*. Podobnie obiekty *A*, *C* oraz *E* zawierają kod, który zakłada istnienie obiektu *D*. Ponieważ obiekt *D* wykorzystuje dwustronną komunikację z pozostałymi obiektami, widać od razu, że został źle zaprojektowany. Przyjmijmy, że możemy dodać następną klasę, aby rozbić powiązania w następujący sposób:



Przenosząc część zadań z obiektu *D* do *X*, zmniejszyliśmy poziom powiązań pomiędzy obiektami w programie. Można założyć, że *X* jest naszym programem głównym, ponieważ zarządza on działaniem pozostałych klas. Najprawdopodobniej obiekt *D* realizował dwa zadania zamiast jednego, więc istnieje związek pomiędzy adekwatnością a powiązaniami.

Minimalizacja liczby powiązań stanowi dobrą praktykę programistyczną. Gdy moduły są powiązane w sposób luźny, znacznie zwiększa się nasza możliwość ich ponownego zastosowania, ponieważ stają się bardziej adekwatne. Warto poświęcić wiele czasu i wysiłku w trakcie projektowania aplikacji. Wysiłek ten opłaci się podczas pracy nad projektem realizującym podobne zadania. Będzie można oszczędzić sporo czasu na tworzeniu komponentów programistycznych realizujących podobne zadania.

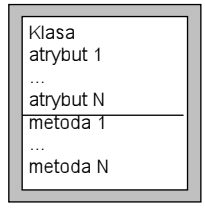
> Pisanie adekwatnych, lecz luźno powiązanych modułów powinno być naszym nadrzędnym celem podczas tworzenia kodu z założenia przeznaczonego do ponownego wykorzystania i rozwoju.

Modelowanie obiektowe z użyciem UML

Wszystkie języki wymagają pewnej formalizacji opisu. Diagramy klas utworzone w UML-u (*Unified Modeling Language*) dają nam możliwość opisu projektów za pomocą symboli graficzno-tekstowych zamiast fragmentów kodu. Język UML jest określany jako *meta-język*. Został opracowany przez *Object Management Group* (*http://www.omg.org/*) i jest przeznaczony do opisu różnych faz procesu projektowania i tworzenia oprogramowania. UML bardzo dobrze sprawdza się jako sposób opisu projektów baz danych oraz programów zorientowanych obiektowo.

Zastosowanie UML-u jest doskonałym sposobem realizacji projektu aplikacji przed rozpoczęciem pisania kodu. Etap ten umożliwia spojrzenie na poszczególne elementy projektu w sposób niezależny od zastosowanego języka programowania, przypisanie programistom poszczególnych elementów systemu, a także oszacowanie czasu trwania cyklu tworzenia aplikacji. W tym podrozdziale omówimy kilka kluczowych schematów projektowych, które pomogą w tworzeniu elastycznych i skalowalnych aplikacji internetowych. Zapoznamy się z jednym z elementów UML-u, nazywanym **diagramem klas** używanym do opisu modelu obiektowego.

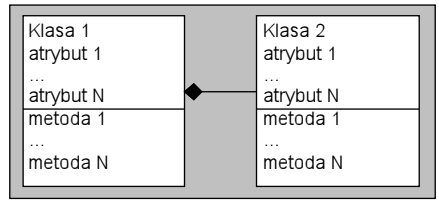
Klasa jest przedstawiana w UML-u w postaci prostokąta zawierającego nazwę, atrybuty i usługi oddzielone liniami. Podstawowy symbol klasy w UML-u wygląda następująco:



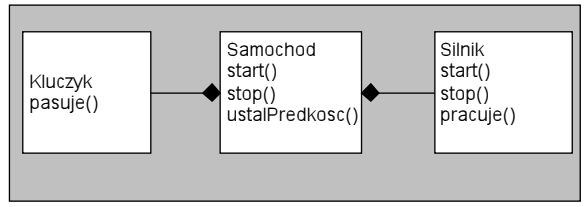
Często atrybuty są pomijane, a wyliczone zostają tylko metody. To zależy od tego, jak dużo czasu potrzebujesz na zapisanie swoich koncepcji.

> Wyliczenie atrybutów wraz z metodami jest dobrą praktyką, szczególnie w przypadku współpracy z bazami danych.

Teraz, gdy znamy sposób modelowania prostych klas, przyjrzyjmy się sposobowi modelowania bardziej skomplikowanych klas zawierających inne klasy. Nie przedstawiamy ich jako atrybuty klasy. Przedstawiamy je jako dwa diagramy klas w UML-u i łączymy je linią z symbolem równoległoboku w następujący sposób:

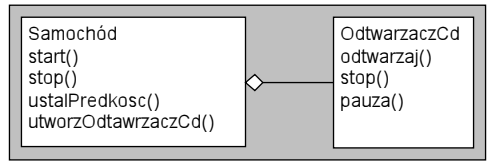


Ten diagram przedstawia, że `Klasa 2` jest zagnieżdżona w `Klasa 1`. Nasz przykład klasy `Samochod` można zobrazować następująco:



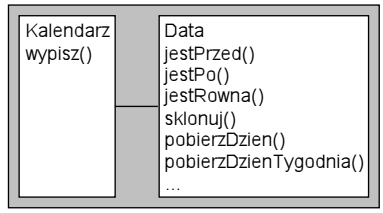
Oznacza to, że `Samochod` zagnieżdża w sobie obiekty klas `Stacyjka` oraz `Silnik`. Wypełnione równoległoboki oznaczają, że komponenty muszą zostać utworzone w celu prawidłowego funkcjonowania obiektu, w którym są zagnieżdżone. Pamiętajmy, że za prawidłową inicjację wszelkich atrybutów klasy odpowiedzialne są konstruktory. Analizując powyższy diagram, programista widzi, jakie obiekty muszą zostać utworzone w celu udostępnienia wszelkich usług obiektu nadrzędnego.

Możliwe jest również zastosowanie symbolu pustego (niewypełnionego) równoległoboku w diagramie powiązań obiektów. Zastosowanie tego symbolu wskazuje, że nie ma konieczności inicjacji obiektu w celu prawidłowego funkcjonowania klasy nadrzędnej. Oto przykład dodania obiektu klasy `OdtwarzaczCd` do klasy `Samochod`:

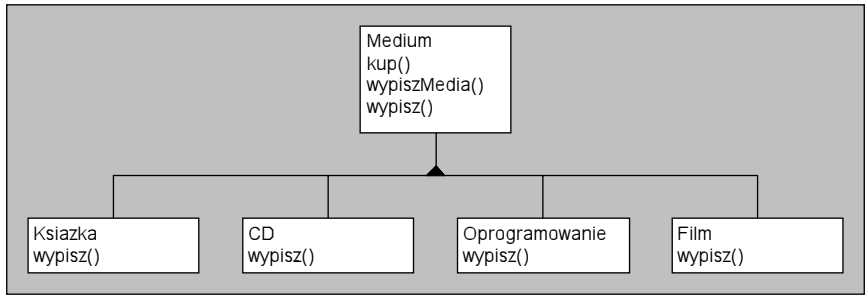


Nie wszystkie samochody muszą mieć zainstalowane odtwarzacze CD, więc sygnalizujemy ten fakt, stosując symbol pustego równoległoboku. W przypadku występowania obiektów, które nie muszą być inicjowane w konstruktorze klasy nadrzędnej, należy zastosować metodę fabryczną (omówioną wcześniej) służącą do utworzenia takiego obiektu, na przykład `utworzOdtwarzaczCd()` w przypadku naszej klasy `Samochod`. Różnica w stosunku do klas `Silnik` oraz `Stacyjka` polega na tym, że nie udostępniamy metod fabrycznych `utworzSilnik()` czy też `utworzStacyjke()` w klasie `Samochod`, ponieważ nie mają one sensu w sytuacji, gdy obiekty wspomnianych klas są inicjowane w konstruktorze.

Zdarza się, że klasa „wykorzystuje" obiekty innych klas w swoich metodach, lecz ich nie zagnieżdża. Na przykład możemy potrzebować klasy `Data` w celu wykonywania operacji na znacznikach czasu systemu UNIX. Taki związek pomiędzy obiektami oznaczany jest w postaci linii bez symbolu równoległoboku.



Skoro poznaliśmy sposób modelowania zagnieżdżania obiektów, przejdźmy do sposobu modelowania dziedziczenia. W naszym przykładzie sklepu z multimediami mieliśmy do czynienia z różnymi rodzajami mediów, na przykład z książkami, płytami CD czy też filmami. Utworzyliśmy klasę macierzystą o nazwie `Medium`, a następnie utworzyliśmy jej klasy potomne w celu reprezentacji różnych typów mediów dostępnych w naszym sklepie. Aby przedstawić związki dziedziczenia pomiędzy klasami `Ksiazka`, `Cd` i innymi a klasą `Medium`, użyjemy symbolu wypełnionego trójkąta łączącego za pomocą linii klasę macierzystą z klasami potomnymi. Oto model w UML-u dla naszego sklepu z multimediami:



Warto zwrócić uwagę, że nie ma potrzeby wyliczania w diagramach klas potomnych wszystkich metod klasy macierzystej. Z założenia wiadomo, że wszystkie klasy potomne posiadają metody klasy macierzystej. Dobrą praktyką jest jednak umieszczanie nazw metod klasy macierzystej przesłoniętych w klasach potomnych, jak metoda `wypisz()` z naszego przykładu.

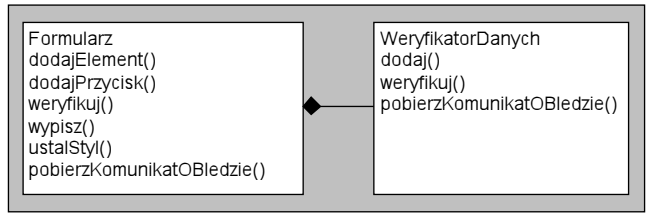
Gdy klasy rozrastają się, można również pominąć je dla oszczędzenia miejsca. Oczywiście, można również zastosować kolejne dziedziczenie, na przykład klasa `Film` może mieć klasy potomne o nazwach `VHS` oraz `DVD`, o ile taka decyzja stanowi właściwe rozwiązanie upraszczające implementację. W dalszej części rozdziału zajmiemy się analizą przypadków, w których lepiej zastosować dziedziczenie oraz innych, które są lepiej reprezentowane w postaci zagnieżdżania obiektów. W tej chwili jednak zajmijmy się specyficznym typem zawierania, nazywanym delegacją.

Delegacja

Delegacja jest specyficzną odmianą zawierania się obiektów służącą ponownemu wykorzystaniu kodu obiektów. Gdy klasa ma za zadanie udostępnienie usługi, może po prostu

oddelegować jej realizację obiektowi, który jest w niej zawarty i po prostu przekazać wynik. Używając delegacji, obiekt nie posiada fizycznego powiązania, jak na przykład silnik z samochodem. Jedynym zadaniem obiektu zawartego w obiekcie nadrzędnym jest udostępnienie usługi w celu uproszczenia projektu obiektu nadrzędnego i uczynienia go bardziej spójnym.

Jako przykład utworzymy obiekt formularza, który wypisuje i sprawdza poprawność danych formularzy WWW. Jednym z projektów mogłoby być sprawdzanie danych wewnątrz obiektu formularza. Jednakże nie jest to najlepsze rozwiązanie. O wiele lepiej byłoby utworzyć osobną klasę obiektu sprawdzającego poprawność i wykorzystać go do udostępnienia metod sprawdzania poprawności danych formularza, w którym obiekt kontrolny jest zawarty. Oto diagram w UML-u ilustrujący ten przykład:



Obiekt klasy `WeryfikatorDanych` jest zawarty w obiekcie klasy `Formularz` i jest tworzony w trakcie jego tworzenia. Dlaczego postępujemy w ten sposób? Załóżmy, że potrzebujemy obiektu weryfikującego dane w innej aplikacji, na przykład w programie e-mailowym. W innym przypadku konieczne byłoby ponowne zakodowanie mechanizmów klasy `WeryfikatorDanych` w każdej nowej aplikacji. Lepiej jest utworzyć klasę `WeryfikatorDanych`, którą można wykorzystać w każdej chwili, gdy wyda się potrzebna.

Zgodnie z naszą wcześniejszą dyskusją dotyczącą adekwatności, klasa `Formularz` zachowuje się w sposób kojarzący się z klasą boską, obsługując elementy formularza, kod JavaScript, wypisywanie formularza z uwzględnieniem formatowania i weryfikacji danych. Rozsądniej byłoby umieścić obsługę tych zadań w osobnych klasach i po prostu delegować zadania klasy `Formularz`.

Kolejną ważną obserwacją jest to, że klasa `Formularz` jest świadoma istnienia klasy `WeryfikatorDanych`, lecz klasa `WeryfikatorDanych` nie wie nic o tym, że jest zawarta w klasie `Formularz`. Dzieje się tak, ponieważ klasa `WeryfikatorDanych` nie posiada żadnego odwołania do klasy `Formularz`. Dzięki temu komunikacja pomiędzy klasami jest jednokierunkowa zamiast dwukierunkowej. A zatem wykorzystując możliwości delegacji, zmniejszyliśmy liczbę powiązań obiektów. Bardzo ważne jest poznanie możliwości ponownego wykorzystania kodu, ponieważ celem programisty powinno być tworzenie adekwatnych obiektów, które są ze sobą bardzo luźno powiązane. Przeanalizujmy poniższy przykład.

Przykład prezentuje trzy metody delegujące zadania do klasy `WeryfikatorDanych`. Pierwszą metodą jest `dodajElement()`, która zgłasza obiektowi klasy `WeryfikatorDanych` wartość, typ oraz komunikat o błędzie dotyczące elementu formularza.

```
/*-----------------------------------------------------------------------
Dodaje nowy element do formularza i zgłasza go weryfikatorowi
-----------------------------------------------------------------------*/
function dodajElement($element) {
    $this->elementy[] = $element;
```

```
// zgłoszenie elementu w weryfikatorze
    $potrzebnaWeryfikacja = isset($element->wyrazenieRegularne) &&
                            isset ($element->komunikatOBledzie);
    if ($potrzebnaWeryfikacja) {
        $this->weryfikator->add($element->wartosc,
        $element->wyrazenieRegularne, $element->komunikatOBledzie);
    }
}
```

Zadaniem metody `dodajElement()` jest dodawanie elementów do formularza, jak również zgłaszanie ich do weryfikacji. Innym sposobem mogłoby być utworzenie dodatkowej listy obiektów do weryfikacji w ramach klasy `Formularz`, lecz zdecydowaliśmy się na przekazanie tego zadania obiektowi `$weryfikator`. Przyjrzyjmy się następnej metodzie przekazującej część zadań weryfikatorowi:

```
/*-----------------------------------------------------------------------
Weryfikuje obiekty formularza. Funkcja zwraca wartość true, jeśli wszystkie
elementy są poprawne, w przeciwnym wypadku zwraca false
-----------------------------------------------------------------------*/
function weryfikuj($element) {
    return $this->weryfikator->weryfikuj();
}
```

Powyższa funkcja przekazuje zadanie weryfikacji obiektów formularza metodzie `weryfikuj()` klasy `WeryfikatorDanych`. Operacja ta spowoduje przechowanie komunikatów o błędach, które można wydobyć za pomocą następującej metody:

```
/*-----------------------------------------------------------------------
Wydobywa komunikaty o błędach wygenerowane podczas wykonania metody
weryfikuj(). Powinna być wywoływana wyłącznie w przypadku zwrócenia
wartości false przez metodę weryfikuj()
-----------------------------------------------------------------------*/
function pobierzKomunikatyOBledach() {
    return $this->weryfikator->pobierzKomunikatyOBledach();
}
```

Udostępniając interfejs na podstawie obiektu weryfikatora, obiekt klasy `Formularz` działa jak obiekt pośredni przekazujący wykonywanie zadań swoim obiektom składowym.

Jak widać zatem, możemy tworzyć eleganckie rozwiązania częstych problemów, stosując delegację usług do wyspecjalizowanych obiektów, i w ten sposób tworząc moduły lepiej nadające się do ponownego wykorzystania i dalszego rozwoju.

Analiza i decyzje projektowe

Obiekty są często projektowane w sposób sugerujący, że ich głównym zadaniem jest przechowywanie danych. Taka metodologia jest nieprawidłowa w programowaniu zorientowanym obiektowo. Cechą różniącą obiekt od innych struktur danych jest umiejętność świadczenia usług specyficznych dla obiektu. Przyjrzyjmy się przykładowi, w którym obiekt nie udostępnia usług, których można byłoby od niego oczekiwać:

```
class Punkt {
    var $x;
    var $y;
    var $kolor;
    function Punkt($x, $y) {
        $this->ustawX{$x);
        $this->ustawY($y);
    }
    function ustawX($x) {
        $this->x = $x;
    }
    function ustawY($y) {
        $this->y = $y;
    }
    function ustawKolor($kolor) {
        $this->kolor = $kolor;
    }
    function pobierzX() {
        return $this->x;
    }
    function pobierzY() {
        return $this->y;
    }
    function pobierzKolor() {
        return $this->kolor;
    }
    function rysuj() {
        ...
    }
}
```

Na pierwszy rzut oka wygląda to na niezłą reprezentację punktu na płaszczyźnie. Przyjrzyjmy się jednak możliwościom zastosowania takiej klasy:

```
$punkt = new Punkt(0, 40);
$punkt->rysuj();
$x = $punkt->pobierzX();
$x += 32;
$y = $punkt->pobierzY();
$y += 96;
$punkt->ustawX($x);
$punkt->ustawY($y);
$punkt->rysuj();
```

Zastanówmy się przez chwilę, jakie problemy mogą pojawić się z wykorzystaniem takiego kodu. Mimo że pozornie wszystko jest w porządku, po głębszym zastanowieniu można wykryć problemy. Dlaczego nasz kod podejmuje decyzje w imieniu obiektu? Dlaczego stosujemy operacje typu `$x = $x + 32`, skoro powinien robić to sam obiekt? Czy obiekt nie powinien umieć dokonywać operacji na swoich atrybutach? Przykładowa klasa nie została zdefiniowana prawidłowo, ponieważ nie wykorzystuje osadzania. Poniżej przedstawiamy lepszą definicję klasy `Punkt`:

```
class Punkt {
    var $x;
    var $y;
    var $kolor;
```

```
    function Punkt($x =0, $y = 0) {
        $this->przesunDo($x, $y);
    }
    function przesunDo($x, $y) {
        $this->x = $x;
        $this->y = $y;
    }
    function przesunX($wartosc) {
        $this->x += $wartosc;
    }
    function przesunY($wartosc) {
    $this->y += $ wartosc;
    }
    function przesun($wartoscX, $wartoscY) {
        $this->przesunX($wartoscX);
        $this->przesunY($wartoscY);
    }
    function ustawKolor($kolor) {
        $this->kolor = $kolor;
    }
    function rysuj() {
        ...
    }
}
```

Zobaczmy, jak bardzo uprościł się nasz kod i w jaki sposób osiągnęliśmy hermetyzację obiektu. Wykorzystanie klasy jest podobne jak w poprzednim przykładzie:

```
$punkt = new Punkt(0, 40);
$punkt->przesun(32, 96);
$punkt->rysuj();
```

Szczegóły implementacji zostały ukryte, a obiekt podejmuje samodzielne decyzje. Stosując się do podobnych zasad zwiększających hermetyzację wszystkich obiektów w programie, osiągamy większą czytelność i możliwość łatwiejszego dokonywania zmian w programie.

Funkcje PHP obsługujące klasy

PHP udostępnia szereg funkcji upraszczających pracę z obiektami i klasami. Niektóre z tych funkcji pozwalają uniknąć niektórych problemów w przypadku konieczności wykorzystywania kiepskich projektów obiektowych.

get_class()

```
string get_class(object ob)
```

Funkcja `get_class()` zwraca nazwę klasy obiektu. Szczególnie użyteczna jest podczas procesu wyszukiwania i usuwania błędów, umożliwiając sprawdzenie, czy w programie biorą udział właściwe obiekty. Na przykład, gdy posiadamy metodę oczekującą obiektu klasy `Uzytkownik`, możemy zastosować tę funkcję w celu usprawnienia wyszukiwania błędów:

```
function autoryzacja($uzytkownik) {
    assert(get_class($uzytkownik) == 'uzytkownik');
    if ($user->wydzial == $this->wymaganyWydzial) {
        return true;
    }
    return false;
}
```

Funkcja `get_class()` umożliwia weryfikację poprawności danych i jest użyteczną alternatywą funkcji `is_object()` sprawdzającej, czy argument jest obiektem klasy. Taka możliwość pozwala na zaoszczędzenie czasu podczas usuwania błędów w aplikacjach wykorzystujących wiele powiązań pomiędzy obiektami różnych klas.

Należy zwrócić uwagę na fakt, że PHP zamienia nazwy klas na małe litery, należy więc w porównaniach stosować nazwy klas składające się z małych liter. W naszym przykładzie porównywaliśmy wynik działania funkcji `get_class()` z napisem `uzytkownik` zamiast `Uzytkownik`. Porównanie z napisem `Uzytkownik` dałoby wynik negatywny, ponadto wywołałoby błąd asercji.

get_parent_class()

```
string get_parent_class(object ob)
```

Funkcja ta jest szczególnie użyteczna podczas sprawdzania poprawności kodu wykorzystującego mechanizmy polimorfizmu. Nie będzie potrzebna w ostatecznej wersji aplikacji, ponieważ w tym przypadku wszystkie obiekty powinny być egzemplarzami klas potomnych prawidłowo skomponowanych klas macierzystych. Jeśli założenie takie nie jest spełnione, tworzona aplikacja zawiera poważne błędy koncepcyjne. Oto fragment kodu kontrolnego w metodzie polimorficznej:

```
function wypiszWszystko() {
    foreach ($this->elementy as  $element) {
        assert(get_parent_class($element) == 'element');
        $element->display();
    }
    return false;
}
```

Funkcja oczekuje tablicy elementów będących egzemplarzami klas potomnych klasy `Element` i w kolejności wypisuje je na wyjście. Program stosuje asercję zakładającą przynależność każdego elementu do klasy potomnej klasy `Element`. Kod podobny do powyższego może okazać się użytecznym w trakcie usuwania błędów, lecz powinien być wyłączany w środowisku produkcyjnym. Więcej informacji na temat usuwania błędów znajdziesz w rozdziale 6.

Ograniczenia PHP

Jak wspomnieliśmy wcześniej w tym rozdziale, implementacja technik zorientowanych obiektowo w PHP posiada swoje ograniczenia. W tym podrozdziale omówimy najczęściej wspominane ograniczenia PHP, takie jak brak atrybutów statycznych, brak destruktorów oraz brak wielokrotnego dziedziczenia.

Brak atrybutów statycznych

PHP udostępnia programistom operator wywołania metody klasy (wspomniany w podrozdziale dotyczącym dziedziczenia) umożliwiający statyczne wywoływanie metod. Niestety, język ten nie udostępnia statycznych atrybutów. Czym jest atrybut statyczny? Jest to zmienna globalna przywiązana do przestrzeni nazw klasy (w przeciwieństwie do przestrzeni nazw obiektu). Oznacza to, że jest to pojedyncza zmienna używana przez wszystkie egzemplarze klasy, nie zaś unikalna dla każdego z egzemplarzy.

Do czego są potrzebne atrybuty statyczne? Czasem wygodniej jest używać jednego zestawu danych we wszystkich obiektach tej samej klasy, zamiast stosować je wielokrotnie w każdym egzemplarzu, oszczędzając w ten sposób pamięć. Drugi powód stosowania atrybutów statycznych to kontrola stanu jakiegoś atrybutu, jednolitego dla wszystkich obiektów klasy, jak na przykład liczba obiektów klasy istniejących w danym momencie.

Wiele języków, na przykład Java, obsługuje atrybuty statyczne, jednak PHP nie posiada takiej własności. Można jednak zasymulować tę obsługę, stosując kombinację zmiennych globalnych i metod statycznych. Przyjrzyjmy się zastosowaniu takiej techniki na przykładzie klasy `Jablko` wykorzystującej symulację atrybutu statycznego w celu śledzenia liczby egzemplarzy klasy istniejących w pamięci:

```
<?php
// jablko.php
class Jablko {
var $jestZjedzone;
```

Konstruktor klasy `Jablko` zawiera kod zwiększający o jeden wartość zmiennej globalnej `$liczbaJablek`, co odzwierciedla fakt utworzenia nowego jabłka.

```
    function Jablko() {
        global $liczbaJablek;
        $liczbaJablek++;
        $this->jestZjedzone = false;
    }
```

Z kolei metoda `zjedz()` zmniejsza o jeden wartość zmiennej globalnej `$liczbaJablek`:

```
    function zjedz() {
        if (!$this->jestZjedzone()) {
            global $liczbaJablek;
            $liczbaJablek--;
            $this->jestZjedzone = true;
        }
    }

    function jestZjedzone() {
        return $this->jestZjedzone;
    }
```

Zwróć uwagę, że zmienna globalna zostanie zmniejszona wyłącznie w przypadku, gdy jabłko nie zostało jeszcze zjedzone. Pozwala to na zachowanie spójności emulowanego atrybutu statycznego. Na końcu definiujemy metodę `liczba()` zwracającą liczbę istniejących w pamięci obiektów klasy `Jablko`:

```
    // metoda statyczna
    function liczba() {
        global $liczbaJablek;
        return $liczbaJablek;
    }
}

$j1 = new Jablko(); // ustawia $liczbaJablek na 1
$j2 = new Jablko(); // ustawia $liczbaJablek na 2
$j3 = new Jablko(); // ustawia $liczbaJablek na 3
echo(Jablko::liczba() . "<br>"); // wypisuje 3
$j1->zjedz(); // ustawia $liczbaJablek na 2
$j2->zjedz(); // ustawia $liczbaJablek na 1
$j4 = new Jablko(); // ustawia $liczbaJablek na 2
echo(Jablko::liczba() . "<br>"); // wypisuje 2
?>
```

Mimo że stosowanie zmiennych globalnych jest powszechnie uważane za nieeleganckie, zastosowanie tej metody w celu zasymulowania atrybutu statycznego w PHP jest całkiem efektywne.

Po wywołaniu powyższego kodu przeglądarka wyświetli liczby 3 oraz 2, zgodnie z komentarzami w powyższym kodzie. Emulowanie atrybutów statycznych to użyteczna technika, lecz należy być świadomym problemów związanych z jej stosowaniem:

- technika ta nie zabezpiecza zmiennych przed modyfikacją „z zewnątrz" bez wykorzystania do tego celu metod statycznych, co w konsekwencji może doprowadzić do zaburzenia integralności atrybutu z klasą;
- atrybut nie jest w rzeczywistości związany z klasą, co utrudnia innym programistom orientację i wymaga zastosowania większej liczby komentarzy w celu zaznaczenia wykorzystania atrybutu statycznego;
- inne obiekty mogą bez problemu zamazać (usunąć) zawartość zmiennej globalnej, jeśli nie są świadome wykorzystania jej w charakterze atrybutu statycznego.

Wymienione problemy są typowe dla stosowania zmiennych globalnych, więc symulowanie atrybutów statycznych za pomocą zmiennych globalnych nie stanowi wyjątku.

Brak destruktorów

Konstruktory inicjują stan obiektów tak, aby można było z nich korzystać natychmiast po utworzeniu. Inną koncepcją programowania zorientowanego obiektowo jest pojęcie destruktora, używanego do usuwania atrybutów obiektu, w tym zagnieżdżonych obiektów, lub też wykonania innych czynności związanych z kończeniem działania obiektu, jak na przykład zamykania połączenia z bazą danych. W PHP nie istnieje możliwość likwidowania obiektów w ten sposób. Zamiast tego PHP po prostu usuwa z pamięci obiekty utworzone przez skrypt w momencie zakończenia działania skryptu.

Brak wielokrotnego dziedziczenia

Wielokrotne dziedziczenie umożliwia jednoczesne dziedziczenie atrybutów i metod klasy po kilku klasach macierzystych. Na przykład istnienie klas `Inzynier` oraz `Dyrektor` umożliwia utworzenie klasy potomnej `DyrektorTechniczny` dziedziczącej po obu wspomnianych klasach.

W PHP nie można zdefiniować dziedziczenia atrybutów oraz metod po więcej niż jednej klasie macierzystej za pomocą słowa kluczowego `extends`. W przeciwieństwie do PHP, inne języki, na przykład C++, umożliwiają wielokrotne dziedziczenie. Podobnie jak ma to miejsce w przypadku atrybutów statycznych, możemy emulować pożądane mechanizmy, w tym przypadku stosując kombinację dziedziczenia i delegacji. Sztuczka polega na dziedziczeniu po jednej klasie, natomiast możliwości pozostałych klas osiągniemy, definiując metody delegujące zadania do metod obiektów zagnieżdżonych. Takie rozwiązanie nie jest optymalne w przypadku, gdy liczba klas, po których chcemy dziedziczyć, jest duża, lecz w przypadku dziedziczenia po dwóch, trzech klasach może wydać się dobrym rozwiązaniem.

Oto przykład klasy `DyrektorTechniczny` będącej przykładem zastosowania powyższej techniki symulacji wielokrotnego dziedziczenia po klasach `Inzynier` oraz `Dyrektor`:

```
<?php
// dyrektor.php
class Dyrektor {
    var $imie, $nazwisko;
    function Dyrektor($imie, $nazwisko) {
    $this->imie = $imie;
    $this->nazwisko = $nazwisko;
    }
    function zarzadzajPracownikiem($pracownik) {
        // kod zarządzający pracownikiem
        ...
    }
    function placPracownikowi($pracownik)
    {
        // kod wypłacający pensję pracownikowi
        ...
    }
}
?>
```

Nie ma tu na razie nic nadzwyczajnego. Ta klasa macierzysta zawiera dwie metody umożliwiające dyrektorowi zarządzanie pracownikami oraz wypłacanie im pensji, czyli to, czego nie robią inżynierowie.

```
<?php
// inzynier.php
class Inzynier {
    var $imie, $nazwisko;
    function Inzynier($imie, $nazwisko, $rodzajInzyniera) {
    $this->imie = $imie;
    $this->nazwisko = $nazwisko;
    }
```

```
    function projektuj($projekt) {
        // kod przypisujący projekt inżynierowi
    }

    function pobierzRodzajInzyniera()
    {
        return $this->rodzajInzyniera;
    }
}
?>
```

Klasa `Inzynier` udostępnia metodę definiującą projekty prowadzone przez inżyniera. Chcąc zdefiniować klasę implementującą dyrektora technicznego, oczekujemy, że będzie zarządzał inżynierami, zatwierdzał ich listy płac i rozdzielał im projekty. Zastosujmy więc naszą sztuczkę z dziedziczeniem i delegacją, aby zasymulować wielokrotne dziedziczenie.

```
<?php
class DyrektorTechniczny extends Dyrektor {
    var $inzynier
```

Klasa `DyrektorTechniczny` dziedziczy po klasie `Dyrektor`, więc zawiera te same atrybuty oraz metody co klasa macierzysta. Aby udostępnić mechanizmy klasy `Inzynier`, zagnieździmy klasę `Inzynier` wewnątrz klasy `DyrektorTechniczny`, a w konstruktorze umieścimy inicjację tego obiektu:

```
    function DyrektorTechniczny($imie, $nazwisko, $rodzajInzyniera) {
        Dyrektor::Dyrektor($imie, $nazwisko);
        $this->inzynier = new Inzynier($imie, $nazwisko, $rodzajInzyniera);
    }
    function projektuj($projekt) {
        $this->inzynier->projektuj($projekt);
    }
    function pobierzRodzajInzyniera() {
        return $this->inzynier->pobierzTypInzyniera();
    }
}
$dyrektorTechniczny = new DyrektorTechniczny('Jan', 'Kowalski', 'Mechanika');
?>
```

Takie rozwiązanie zapewnia wystąpienie obiektu klasy `Inzynier` w każdym obiekcie klasy `DyrektorTechniczny`. Aby udostępnić metodę `projektuj()`, po prostu delegujemy ją do osadzonego obiektu `$inzynier`:

```
function projektuj($projekt) {
    $this->inzynier->projektuj($projekt)
}
```

Aby udostępnić metodę `pobierzRodzajInzyniera()`, postępujemy podobnie:

```
function pobierzRodzajInzyniera() {
    return $this->inzynier->pobierzRodzajInzyniera();
}
```

Dzięki temu każdy egzemplarz klasy `DyrektorTechniczny` będzie udostępniać metody obu klas. Zastosowanie przedstawionego sposobu znacznie się komplikuje, gdy chcemy jednocześnie dziedziczyć po większej liczbie klas, dlatego bardzo trudno byłoby tworzyć nowe

klasy i dziedziczyć po różnych kombinacjach klas. Podana przez nas metoda powinna być stosowana z rozwagą, ponieważ mamy nadzieję, że w przyszłych wersjach PHP opcja wielokrotnego dziedziczenia będzie już dostępna.

> Podana metoda może pomóc w rozwiązaniu niektórych problemów, lecz wraz ze wzrostem liczby klas, po których chcemy dziedziczyć, kod może stać się mniej czytelny i trudniejszy w dalszym rozwoju.

Modelowanie złożonego komponentu WWW

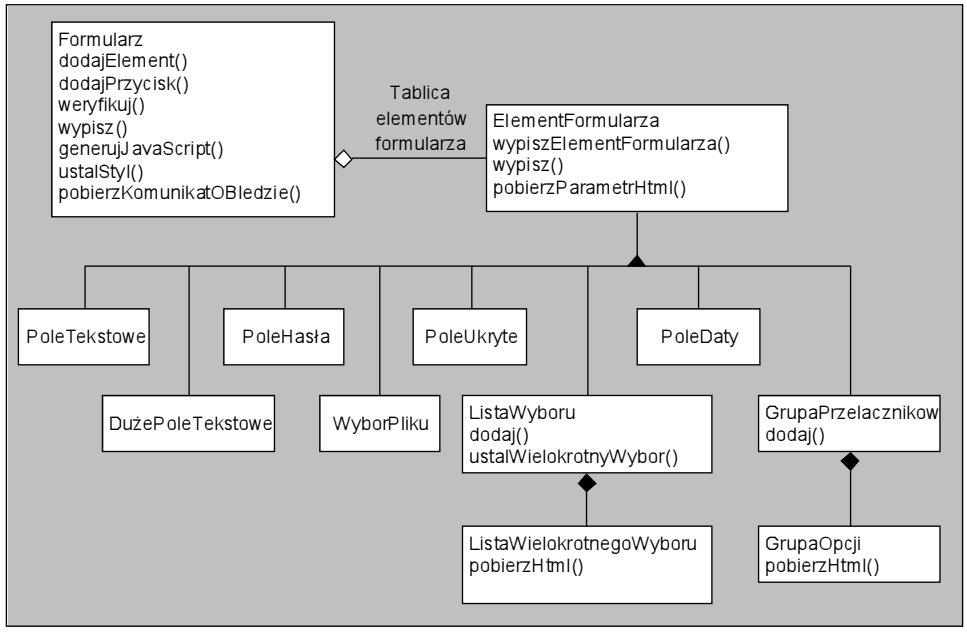
W tym podrozdziale zaprojektujemy mechanizm omówionego wcześniej formularza WWW. Przy tej okazji przekażemy kilka dodatkowych sposobów realizacji struktury obiektów oraz kilka układów projektowych, które mogą być użyteczne w realizowanych projektach. Zdefiniujmy wymagania dotyczące naszego mechanizmu. Mechanizm ten powinien:

- umożliwiać tworzenie kilku formularzy na stronie;
- umożliwiać zmianę wyglądu formularzy bez modyfikacji logiki formularza;
- udostępniać jednolity interfejs służący do dodawania elementów formularza oraz przycisków;
- udostępniać weryfikację danych po stronie użytkownika (z użyciem JavaScriptu) oraz weryfikację po stronie serwera (z użyciem wyrażeń regularnych);
- udostępniać kilka standardowych definicji weryfikacji najczęściej spotykanych danych, na przykład adresów e-mail;
- umożliwiać wypisywanie etykiet wymaganych pól pogrubioną czcionką;
- umożliwiać ponowne wypisanie formularza z zaznaczeniem błędnie wypełnionych elementów;
- automatycznie wypisywać błędy, bez formatowania;
- wykonywać wszystkie wymienione wyżej zadania za pomocą pojedynczego skryptu PHP.

Na pierwszy rzut oka może wydawać się, że realizacja powyższych zadań wymaga utworzenia skomplikowanego modułu. Jednak dzięki odpowiedniemu projektowi architektury możemy osiągnąć zadowalające wyniki bez wprowadzania zbędnej komplikacji. W rzeczywistości istnieje bardziej elastyczny projekt mechanizmu formularzy, niż ten, który chcemy zademonstrować, jednakże wymaga on zastosowania wielu klas zewnętrznych. Celem naszego przykładu jest zademonstrowanie wszystkich koncepcji, które przedstawiliśmy do tej pory. Chcemy pokazać, w jaki sposób można rozwiązać problem utworzenia mechanizmu formularza WWW z zastosowaniem obiektów.

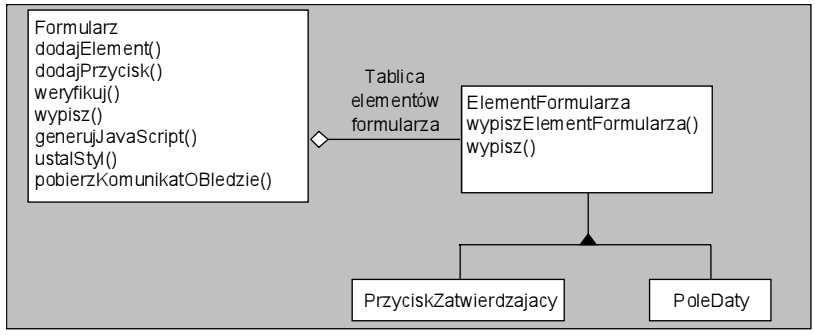
Na początku należy zdefiniować klasę `Formularz`, ponieważ to ona jest główną klasą w naszym mechanizmie. `Formularz` powinien umieć wypisać swoją zawartość, używając definicji stylu, dodawać elementy i przyciski, tworzyć kod weryfikujący dane w JavaScripcie działający po stronie klienta, weryfikować swoje dane po stronie serwera, a także wypisywać komunikat o błędzie w przypadku problemów z weryfikacją danych.

Następnie potrzebna nam będzie struktura obiektów modelująca wszelkie możliwe elementy formularzy oraz przyciski. Zastosujemy klasę macierzystą o nazwie `ElementFormularza` i rozszerzymy ją za pomocą dziedziczenia tak, aby obejmowała wszystkie elementy formularza. Takimi elementami mogą być pola tekstowe, pola daty, duże pola tekstowe, hasła, pola wysyłki plików czy też listy wielokrotnego wyboru. Tworząc klasę macierzystą `ElementFormularza` i wykorzystując możliwości polimorfizmu, możemy dodawać dowolną liczbę typów elementów formularza bez konieczności modyfikacji logiki definiującej wypisywanie, weryfikację oraz generowanie kodu w JavaScripcie. Przyjrzyjmy się naszemu modelowi obiektowemu:



Ten model wykorzystuje niemal tę samą standardową hierarchię zagnieżdżania i dziedziczenia, którą omówiliśmy przy okazji omawiania wcześniejszych przykładów w tym rozdziale, z dwoma wyjątkami. Po pierwsze, schemat pokazuje, że elementy formularza nie są zawarte w obiekcie formularza. Chociaż fizycznie elementy są składowymi formularza, zachowują się inaczej w momencie jego utworzenia. Dodaliśmy metody `dodajElement()` oraz `dodajPrzycisk()` służące do dodawania elementów do formularza. Formularz nie posiadający elementów jest nadal pełnoprawnym, choć niezbyt użytecznym obiektem klasy.

Druga komplikacja dotyczy związków zawierania oraz metody `dodaj()` w klasach `ListaWyboru` oraz `ListaWielokrotnegoWyboru`. Pozwala to na dodawanie elementów do pola wyboru w ten sam sposób, w jaki można to zrobić w przypadku formularza. Mechanizm ten jest praktycznie przezroczysty dla programisty. Znaczenie naszego kodu polega na tym, że metoda `wypisz()` każdej z klas potomnych klasy `ElementFormularza` może zostać zrealizowana w dowolny sposób. Możemy wypisać znaczniki elementów pola wyboru w dowolny sposób, nie naruszając żadnej z zasad opisanych w modelu dziedziczenia. Mniejsze, lecz podobne modele dziedziczenia możemy zdefiniować dla przycisków:



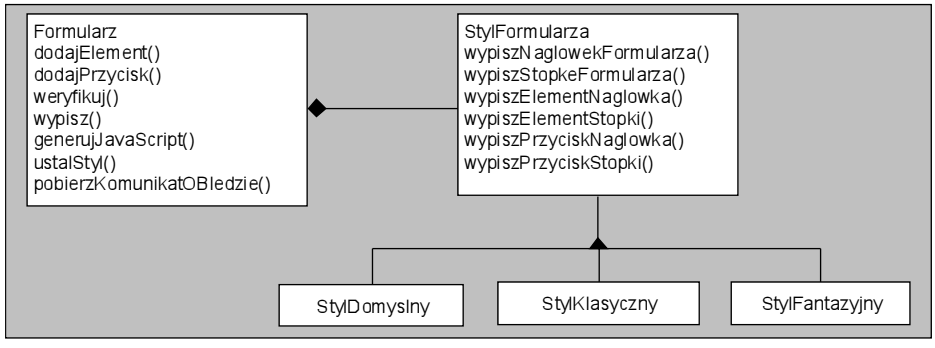
Posiadamy już solidny fundament dla struktur formularzy, lecz w jaki sposób zadbać o ich atrakcyjny wygląd? Zastosujemy osobną klasę `StylFormularza`, co pozwoli nam na dowolne definiowanie stylu formularza. W naszym przypadku zdecydowaliśmy się na wykorzystanie HTML-u. Często logika prezentacji jest umieszczana w tym samym miejscu, co logika aplikacji, lecz nie jest to dobra praktyka. Komponenty stylu lub elementy dekoracyjne pozwalają na umieszczenie logiki prezentacji osobno, dzięki czemu nasz obiekt jest bardziej spójny. Dodatkową korzyścią płynącą z oddzielenia elementów dekoracyjnych od logiki obiektu jest to, że można zmienić sposób prezentacji obiektu bez zmiany logiki obiektu, dzięki czemu łatwiej go rozbudowywać i stosować w nowym projekcie.

W celu zdefiniowania stylu formularzy wykorzystamy możliwość delegacji. Jednym ze stosowanych układów projektowych jest zastosowanie zewnętrznego obiektu dekoratora. Dzięki strukturalnej naturze języka HTML łatwo jest zrealizować taką funkcję za pomocą obiektu osadzonego. Dzięki temu diagram przepływu informacji pomiędzy obiektami jest o wiele czytelniejszy bez negatywnych efektów ubocznych, ponieważ możemy osiągnąć te same możliwości, co w przypadku zastosowania zewnętrznego obiektu dekoratora.

Klasa `StylFormularza` została rozszerzona o trzy klasy potomne. Klasa `StylDomyslny` jest wykorzystywana przez konstruktor klasy `Formularz`. Pozostałe dwie klasy reprezentują dowolne style, które można zastąpić własnymi, w zależności od upodobań. Dzięki takiemu projektowi możemy tworzyć efektowne formularze, definiując nowe obiekty stylu.

```
$formularz->ustawStyl(new StylFantazyjny());
$formularz->wypisz();
```

Oto diagram w UML-u przedstawiający formularz z komponentem definicji stylu:

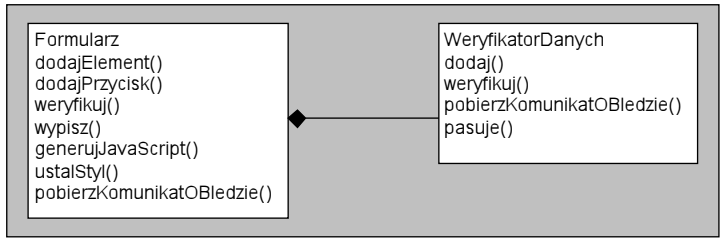


Jeśli chcemy zmodyfikować kolorystykę, możemy powyższy zapis zmodyfikować następująco:

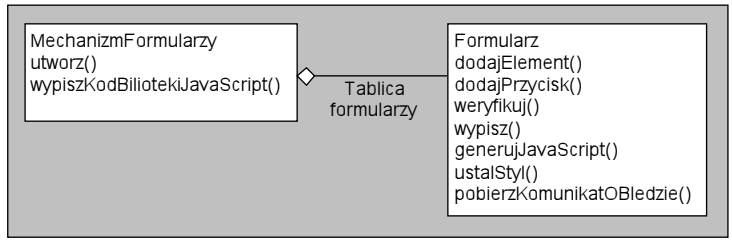
```
$formularz->ustawStyl(new NowyStyl());
$formularz->wypisz();
```

Dzięki temu nie ma potrzeby przeprowadzania jakiejkolwiek modyfikacji w ramach logiki aplikacji.

Przedostatni element stanowi obiekt weryfikacji danych formularza. Omawiając delegację, stworzyliśmy już odpowiednią klasę. Model obiektowy tej klasy przedstawiamy na kolejnym diagramie:



Aby zarządzać wieloma formularzami, potrzebujmy wspomnianej wcześniej klasy `MechanizmFormularzy`, która jest naszą ostatnią klasą. `MechanizmFormularzy` będzie narzędziem służącym do zarządzania większą liczbą formularzy i posłuży do odtwarzania kodu JavaScript. Klasa ta działa również jako klasa fabryczna, ponieważ możemy tworzyć formularze za jej pomocą. Oto diagram przedstawiający model obiektowy klasy `MechanizmFormularzy`:



Powodem, dla którego obiekt klasy `MechanizmFormularzy` tworzy formularze, jest możliwość kontroli przezeń zależności określających kod JavaScript niezbędny do realizacji zadań formularzy. Jest to potrzebne dla implementacji mechanizmu weryfikacji poprawności danych po stronie klienta. W tym momencie nasz mechanizm jest już prawie ukończony. Poniżej przedstawiamy kod demonstrujący łatwość wykorzystania naszego modelu w celu utworzenia formularza, który przedstawiliśmy wcześniej:

```
$mechanizmFormularza = new MechanizmFormularza();
$formularz = $mechanizmFormularza->utworz('formularz', 'Nazwa Formularza',
                                          $PHP_SELF, 'post');
$formularz->dodajElement(new NaglowekFormularza('Informacje ogólne'));
$formularz->dodajElement(new PoleTekstowe('imie_nazwisko', '',
                         'Imię i nazwisko', ALPHA,
                         'Brak imienia i nazwiska', true));
$formularz->dodajElement(new PoleUkryte('idUzytkownika', '1'));
```

```
$formularz->dodajElement(new PoleHasla('haslo', '', 'Hasło', PASSWORD,
                        'Hasło musi składać się co najmniej z czterech znaków',
                                        true));
$formularz->dodajElement(new PoleTekstowe('email1, '', 'Adres e-mail', EMAIL,
                                          'Nieprawidłowy adres e-mail',
                                          true));
$formularz->dodajElement(new DuzePoleTekstowe('opis', '', 'Opis', ALPHANUMERIC,
                                              'Brak opisu osoby', false,
                                              array('rows' => 10,
                                                    'cols' => 40,)));
$formularz->dodajElement(new PoleDaty('data_zatrudnienia', '',
                                      'Data zatrudnienia', false));
$formularz->dodajElement(new WyborPliku('plik', 'Plik', false));
$wybor = new ListaWyboru('rodzaj_stawki', 'S', 'Rodzaj stawki', true);
$wybor->add('Godzinowa', 'G');
$wybor->add('Miesięczna', 'M');
$formularz->dodajElement(Swybor);
$wybor = new ListaWielokrotnegoWyboru('stanowisko', $stanowisko, 'Stanowisko',
                                      false);
$wybor->add('Planista', 'P');
$wybor->add('Dyrektor', 'D');
$wybor->add('Inżynier', 'I');
$wybor->add('Analityk', 'A')}
$formularz->dodajElement($wybor);
$formularz->dodajPrzycisk(new PrzyciskZatwierdzajacy('wyslij', 'Wyślij'));
$formularz->dodajPrzycisk(new PrzyciskZerujacy('wyczysc', 'Wyczyść'));

if ($wyslij == 'Wyślij') {
    $danePoprawne = $formularz->weryfikuj();
    if ($danePoprawne) {
        echo('w porządku');
        // przetwarzanie informacji w bazie danych itp.
    } else {
        echo ($formularz->pobierzKomunikatOBledzie()) ;
        $formularz->wypisz();
        $mechanizmFormularza->wypiszKodBiliotekiJavaScript();
        echo($formularz->generujKodJavaScript());
    }
} else {
    $mechanizmFormularza->wypiszKodBiliotekiJavaScript();
    $formularz->wypisz();
    echo($formularz->generujKodJavaScript());
}
```

Ojej, nie ma ani kawałka kodu implementacji? Co robić? Na to pytanie łatwo odpowiedzieć — napisz to. Jeśli masz zamiar nauczyć się programowania zorientowanego obiektowo i doceniasz koncepcje, które tutaj poznałeś, Twoim pierwszym zadaniem powinno być dokończenie tego modułu. Nie będzie to strata czasu, ponieważ będziesz mógł wykorzystać ten kod w *wielu programach*. Pożegnaj się z tymi wszystkimi darmowymi modułami znajdowanymi w Sieci i zacznij stosować elementy utworzone własnoręcznie.