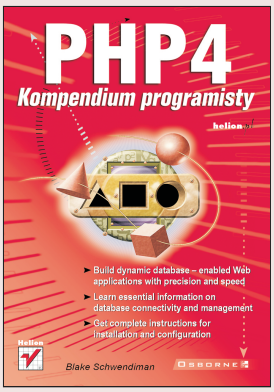
PHP4.
Kompendium programisty



Autor: Blake Schwendiman
Tłumaczenie: Paweł Gonera, Grzegorz Kowalski
ISBN: 83-7197-521-X
Tytuł oryginału: PHP 4 Developer's Guide
Format: B5, stron: 840

Korzystając z tej książki, możesz tworzyć w PHP4 aplikacje WWW korzystające z baz danych. Dowiesz się, jak korzystać z nowych rozszerzeń — sesji, szybszego interpretera oraz rozszerzonej obsługi języków Java i XML — powodujących, że język ten świetnie nadaje się do tworzenia aplikacji WWW. Niniejsza książka zawiera wiele praktycznych przykładów objaśniających sposoby tworzenia witryn za pomocą szablonów. Podręcznik ten jest przydatny dla każdego zawodowego programisty, gdyż opisuje wszystko — od instalacji i konfiguracji, do wykorzystywania istniejących fragmentów oprogramowania.

Znajdziesz tu wszelkie informacje potrzebne do:

- Poznania nowych funkcji PHP4
- Przetwarzania formularzy i kontrolowania poprawności danych
- Oddzielania HTML-a od kodu PHP4
- Uruchamiania i ponownego wykorzystywania kodu
- Korzystania z szablonów i baz danych
- Generowania statycznych stron HTML na podstawie dynamicznych danych

Żadna inna książka nie zawiera tak obszernego opisu programowania w PHP, uwzględniającego najnowsze funkcje, jak ten nieoceniony przewodnik.

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

Spis treści

O Autorze 9

Wstęp 11

Rozdział 1. Kompilacja i instalowanie PHP 15
Wstęp 15
Pobieranie PHP 15
Instalowanie wersji binarnej 16
Binarna instalacja dla Windows 16
Inne instalacje binarne 19
Kompilowanie PHP 20
Kompilowanie PHP w systemach uniksowych 20
Kompilowanie PHP w środowisku Windows 25
Podsumowanie wiadomości na temat kompilacji PHP 27
Konfigurowanie PHP 27
Korzystanie z pliku php.ini 28
Inne metody zmiany konfiguracji PHP 29
Podsumowanie 30

Rozdział 2. Język 31
Wstęp 31
Ogólne informacje na temat składni 31
Typy 32
Liczby — całkowite i zmiennoprzecinkowe 33
Ciągi 33
Tablice 35
Zmienne i stałe 36
Zmienne predefiniowane 37
Zasięg zmiennych 40
Stałe 41
Operatory i kolejność ich występowania 41
Instrukcje sterujące 42
if, else, elseif 43
while 43

do .. while....43
for 44
foreach....45
switch....45
break i continue....47
include i require....49
Funkcje....50
Klasy i programowanie obiektowe....51
Porównywanie wzorców....53
Podsumowanie....53

Rozdział 3. Formularze i cookie....55
Wstęp....55
Obsługa formularzy w PHP....57
Skalarne i wielowartościowe elementy formularza....57
Alternatywne metody odczytywania wartości z formularza....58
Wykorzystanie formularzy do przesyłania plików....62
Wykorzystanie danych....62
Kontrola poprawności danych formularza....63
Kontrola danych za pomocą wyrażeń regularnych....63
Kontrolowanie poprawności za pomocą sprawdzania typów....66
Klasa Validator....66
Cookie....68
Specyfika programowania dla WWW....70
Obsługa nieprawidłowych danych....70
Obsługa i formatowanie wyświetlanych danych....73
Podsumowanie....78

Rozdział 4. Operacje wykonywane na plikach....79
Wstęp....79
Odczytywanie i zapisywanie plików....79
Wykorzystanie gniazd....81
Wykorzystanie potoków....82
Klasa File....83
Podsumowanie....85

Rozdział 5. Wysyłanie plików za pomocą formularza....87
Wstęp....87
Wysyłanie pojedynczego pliku....87
Problemy związane z przesyłaniem plików....90
Przesyłanie wielu plików....91
Bezpieczeństwo....92
Podsumowanie....93

Rozdział 6. Współpraca PHP z bazami danych....95
Wstęp....95
Wprowadzenie....95
Funkcje baz danych....96
MySQL....97
Rozpoczynamy pracę z MySQL....97
Wykorzystanie MySQL....97
ODBC....101
Podstawy ODBC....101
Korzystanie z ODBC....104

PHPLIB ... 106
Przechowywanie danych pochodzących z formularzy ... 107
Wykorzystanie możliwości bazy danych ... 110
Podsumowanie ... 112

Rozdział 7. Sesje i stan aplikacji ... 113
Wstęp ... 113
Podstawy działania sesji ... 113
Mechanizm zarządzania sesjami wbudowany w PHP ... 114
Uruchomienie sesji w PHP ... 115
Przesyłanie identyfikatora sesji bez wykorzystania cookie ... 116
Zapisywanie zmiennych sesji w bazie danych ... 120
Inne funkcje i opcje dotyczące sesji ... 125
Wykorzystanie PHPLIB do obsługi sesji ... 127
Tworzenie własnego mechanizmu obsługi sesji ... 130
Inżynieria programowania a sesje ... 130
Podsumowanie ... 133

Rozdział 8. Uwierzytelnianie ... 135
Wstęp ... 135
Podstawowy system uwierzytelniania w serwerze Apache ... 135
Aktualizacja pliku .htaccess z wykorzystaniem PHP ... 138
Podstawowe uwierzytelnianie za pomocą PHP ... 142
Kompletny system uwierzytelniania oparty na PHP ... 143
Podsumowanie ... 149

Rozdział 9. Niezależność aplikacji od przeglądarki ... 151
Wstęp ... 151
Wprowadzenie ... 151
Wewnętrzne funkcje PHP ... 152
BrowserHawk ... 156
Wykorzystanie informacji na temat przeglądarki ... 161
Podsumowanie ... 164

Rozdział 10. Uruchamianie aplikacji WWW ... 165
Wstęp ... 165
Inżynieria programowania a uruchamianie aplikacji WWW ... 166
Projekt aplikacji ... 166
Definiowanie standardów programowania ... 167
Przegląd oprogramowania ... 167
Testowanie ... 168
Uruchamianie ... 168
Programowanie defensywne ... 169
Dostosowanie mechanizmu obsługi błędów do potrzeb aplikacji ... 175
Zaawansowana obsługa błędów ... 179
Podsumowanie ... 186
Bibliografia ... 186

Rozdział 11. Wielokrotne wykorzystanie kodu ... 187
Wstęp ... 187
Ponowne wykorzystanie kodu a inżynieria programowania ... 187
Ponowne wykorzystanie istniejącego kodu ... 188
PHP ... 188
C/C++ ... 190

Java ... 197
COM ... 201
Inne metody ... 204
Podsumowanie ... 205
Bibliografia ... 205

Rozdział 12. Oddzielanie kodu HTML od PHP ... 207
Wstęp ... 207
Wprowadzenie ... 207
Oddzielanie i integracja modułów kodu z wykorzystaniem wewnętrznych funkcji PHP ... 209
Przesłanki ... 209
Implementacja ... 210
Czego należy unikać? ... 214
Oddzielanie i integrowanie kodów HTML i PHP z wykorzystaniem funkcji PHP — podsumowanie ... 214
Wykorzystanie systemu szablonów ... 215
FastTemplate ... 215
Zaawansowane techniki wykorzystania klasy FastTemplate ... 222
Podsumowanie ... 225
Bibliografia ... 225

Rozdział 13. Ten wspaniały PHP! ... 227
Wstęp ... 227
Wysyłanie do przeglądarki plików innych niż HTML ... 227
Skrypty wspomagające pracę administratora sieci ... 233
WDDX ... 239
Monitorowanie sieci ... 244
Podsumowanie ... 246

Rozdział 14. Witryny tworzone na podstawie szablonów ... 247
Podstawy wykorzystania szablonów ... 247
Zapożyczanie ... 258
Personalizacja witryny ... 261
Obsługa wielu języków w witrynie ... 263
Podsumowanie ... 266

Rozdział 15. Tworzenie witryny opartej na bazie danych ... 267
Wstęp ... 267
Projekt bazy danych ... 267
Zarządzanie danymi aplikacji ... 271
Wyświetlanie danych ... 280
Podsumowanie ... 286

Rozdział 16. Generowanie statycznych stron HTML na podstawie dynamicznych danych ... 287
Wstęp ... 287
Koncepcja ... 288
Generowanie stron statycznych ... 288
Wykorzystanie buforowania ... 288
Wykorzystanie klasy FastTemplate ... 290
Techniki buforowania ... 293
Podsumowanie ... 295

Rozdział 17. Witryny handlu elektronicznego ... 297
Wstęp ... 297
Bezpieczeństwo ... 297
Zastosowanie SSL ... 298
Certyfikaty ... 298
Bezpieczeństwo bazy danych ... 299
Przetwarzanie płatności ... 300
Dostarczanie produktów ... 309
Podsumowanie ... 310

Dodatek A Opis funkcji PHP w porządku alfabetycznym ... 311

Dodatek B Predefiniowane zmienne i stałe PHP ... 775
Zmienne ... 775
Zmienne Apache ... 775
Zmienne środowiska ... 778
Zmienne PHP ... 778
Stałe ... 780

Dodatek C Opcje kompilacji PHP ... 783
Bazy danych ... 783
Handel elektroniczny ... 787
Grafika ... 787
Różne ... 788
Sieć ... 794
Działanie PHP ... 795
Serwer ... 796
Tekst i język ... 797
XML ... 797

Dodatek D Opcje konfiguracji PHP ... 799
Ogólne dyrektywy konfiguracji ... 799
Dyrektywy konfiguracji poczty ... 803
Dyrektywy konfiguracji trybu bezpiecznego ... 804
Dyrektywy konfiguracji debuggera ... 804
Dyrektywy ładowania rozszerzeń ... 804
Dyrektywy konfiguracji MySQL ... 805
Dyrektywy konfiguracji mSQL ... 806
Dyrektywy konfiguracji PostgreSQL ... 806
Dyrektywy konfiguracji Sybase ... 806
Dyrektywy konfiguracji Sybase-CT ... 807
Dyrektywy konfiguracji Informix ... 808
Dyrektywy konfiguracji BC Math ... 809
Dyrektywy konfiguracji możliwości przeglądarek ... 810
Dyrektywy konfiguracji zunifikowanego ODBC ... 810

Dodatek E Zasoby Sieci ... 811

Skorowidz ... 813

10.

Uruchamianie aplikacji WWW

Wstęp

Uruchamianie aplikacji WWW jest, tak jak w przypadku innych typów aplikacji, procesem krytycznym. Problemem może być zdalne uruchamianie programu, szczególnie jeśli nie masz odpowiednich uprawnień do administrowania serwerem WWW. W tym rozdziale zaprezentowane są porady i narzędzia, które mogą usprawnić proces uruchamiania aplikacji. Ponadto w niektórych podrozdziałach przedstawione są zasady inżynierii programowania, ponieważ można uniknąć wielu problemów przy uruchamianiu aplikacji, jeżeli jej projekt jest odpowiednio przygotowany.

Z powodu ogromnego zainteresowania aplikacjami WWW powstało wiele narzędzi (między innymi PHP) do tworzenia takich aplikacji. Niektóre spośród tych narzędzi zostały stworzone na podstawie narzędzi już istniejących, na przykład ASP i JSP, natomiast inne zostały stworzone specjalnie dla potrzeb projektowania interaktywnych aplikacji WWW. Większość tych narzędzi nie spełnia podstawowych zasad inżynierii programowania. Być może wynika to z faktu, że projektowanie aplikacji WWW to zupełnie nowa dziedzina informatyki, a pierwszymi projektantami nowej technologii nie byli doświadczeni programiści. Niezależnie od tych przyczyn, do nowego środowiska programowania należy zaadaptować wszystkie istniejące zasady inżynierii programowania. W poniższym podrozdziale zostały przypomniane zasady inżynierii programowania. Przestrzeganie tych zasad pozwala uniknąć wielu błędów w projektowaniu, dzięki czemu uruchamianie aplikacji przebiega sprawniej i szybciej.

Inżynieria programowania a uruchamianie aplikacji WWW

Jak napisałem w rozdziale 3., „Formularze i cookie", można uniknąć sprawdzania poprawności niektórych danych w przypadku zastosowania lepszego mechanizmu wprowadzania danych. Analogicznie, napisanie lepszego (bardziej defensywnego) kodu znacznie skraca czas uruchamiania aplikacji.

Projekt aplikacji WWW musi być dokładnie przemyślany, podobnie jak projekt każdej innej aplikacji. Tworzenie sprawnej, efektywnej aplikacji wymaga opracowania właściwego projektu, zachowania zgodności ze standardami tworzenia oprogramowania, sprawdzania oprogramowania, testowania modułów oraz ostatecznego uruchomienia aplikacji. Poszczególne czynności są omówione poniżej, w kolejnych sekcjach. Zakładam, że posiadasz ogólną wiedzę na temat inżynierii programowania.

Projekt aplikacji

Przed rozpoczęciem pisania kodu należy określić wstępne założenia aplikacji. W przypadku realizacji dużych projektów takie planowanie może zająć tygodnie lub miesiące. Z kolei wstępne założenia niewielkich projektów mogą zostać zapisane na skrawku papieru w ciągu kilku minut. Zawsze należy przeanalizować wymagania aplikacji, a także opracować wszelkie alternatywne sposoby realizacji zadań aplikacji przed rozpoczęciem pisania kodu. Badania firm TRW i IBM wskazują, że wprowadzenie zmian do aplikacji w początkowym okresie programowania jest 10 do 200 razy tańsze niż wprowadzanie tych samych zmian na końcu całego procesu tworzenia aplikacji (McConnell, 1993).

W zależności od rodzaju projektu, określenie wymaganych funkcji aplikacji może być bardzo pracochłonne. Wstępny podział aplikacji na moduły może uprościć ten proces. W aplikacji WWW modułami takimi mogą być: współpraca z bazą danych, autoryzacja użytkownika, obsługa stanu itd. Po określeniu zadań poszczególnych modułów należy — w razie potrzeby — podzielić je na mniejsze fragmenty i zapisać przeznaczenie każdego fragmentu. W małych aplikacjach dobrą strategią jest podział modułów na pliki kodu bądź klasy obiektowe.

Po zdefiniowaniu zadań aplikacji należy opracować architekturę systemu. Trzeba określić rodzaj stosowanego *systemu zarządzania relacyjną bazą danych* (SZRBD) i przeanalizować wiele innych elementów struktury, np. organizację plików kodu, sposób realizacji ewentualnych zmian itp. Trzeba także rozważyć, czy niektóre moduły należy zakupić, czy też lepiej napisać je samodzielnie. Mimo że PHP może działać na wielu serwerach WWW i platformach systemowych, każda z takich kombinacji posiada indywidualne cechy. Należy poświęcić trochę czasu na określenie, która platforma sprzętowa i serwer WWW będzie najlepiej odpowiadać potrzebom tworzonej aplikacji. Wstępny budżet projektu rzadko jest dobrym powodem wyboru określonej platformy. Wybór bazy danych jest równie istotny, jeżeli aplikacja ma być dynamiczna. Ponadto należy rozważyć, czy serwer WWW i baza danych będą pracowały na tym samym komputerze, czy osobno. W zależności od wielkości i charakteru aplikacji może być to krytyczne zagadnienie.

Po przeanalizowaniu tych wszystkich zagadnień należy określić organizację kodu. Zdefiniuj konwencję nazywania plików i katalogów, co uprości identyfikację kodu. Wymyśl alternatywny plan na wypadek, gdy istnieje duże prawdopodobieństwo zmian. Jeżeli przewidujesz występowanie zmian, napisz aplikację, która będzie lokalizowała zmiany w kilku modułach, a resztę — buforowała. Jest to szczególnie istotne wtedy, gdy przy tworzeniu aplikacji korzystasz z narzędzi pochodzących z innych źródeł lub oprogramowania w wersji beta. Tworzenie zastępników takiego kodu jest łatwe do zaimplementowania i w dłuższym okresie czasu zapewnia łatwiejsze utrzymanie aplikacji.

Na koniec należy zadecydować, które moduły aplikacji zostaną stworzone przy pomocy gotowych narzędzi pochodzących od zewnętrznych dostawców. Dylemat „tworzyć czy kupić" jest dość problematyczny. W zależności od harmonogramu projektu, możesz nie mieć wystarczająco dużo czasu, aby móc dostatecznie przetestować dostępne narzędzia. Jednak możesz również nie mieć wystarczająco dużo czasu na stworzenie ich od podstaw. Aby zmniejszyć wpływ tej decyzji na projekt, można stworzyć własne funkcje pośrednie, ukrywające implementację gotowych narzędzi.

Właściwe zaprojektowanie aplikacji wymaga czasu. W dobrych systemach faza projektowania zajmowała 20 do 30 procent czasu ich tworzenia (McConnell, 1993). Jest to czas przeznaczony na projektowanie wysokiego poziomu, a przecież projektowanie szczegółów implementacji również zabiera sporo czasu.

Definiowanie standardów programowania

Zdefiniowanie standardów programowania ułatwia zarządzanie aplikacjami o dowolnej wielkości. Nawet małe aplikacje programowane przez jednego programistę mogą korzystać ze standardów programowania. Taki standard obejmuje sposoby nazywania, komentowania oraz konwencje układu. Niektóre z nich, na przykład układ kodu, są mniej istotne, ponieważ nowoczesne edytory potrafią przeformatować kod, jednak inne, na przykład konwencje nazw plików i katalogów, mogą mieć ogromne znaczenie w procesie konserwacji kodu[1].

Przegląd oprogramowania

Przegląd oprogramowania umożliwia zrealizowanie kilku celów naraz. Na przykład, przegląd oprogramowania umożliwia porównanie kodu ze standardami kodowania. Początkujący programiści mogą korzystać z wiedzy bardziej doświadczonych kolegów. Przeglądy oprogramowania umożliwiają także zapewnienie odpowiedniej jakości oprogramowania. Analizy przeglądów oprogramowania stosowanych przy tworzeniu rzeczywistych aplikacji wykazały, że przeglądy umożliwiają wykrywanie błędów ze skutecznością 55 – 60%. Ten wynik należy zestawić z jedynie 25-procentową skutecznością testowania modułów, 35-procentową testowania funkcji oraz

[1] Z powodu swobodnego traktowania typów w PHP szczególnie istotne jest odpowiednie nazywanie zmiennych. Dobrym pomysłem może być stosowanie tzw. notacji węgierskiej, w której nazwa zmiennej zaczyna się od liter określających jej typ, np.: `nIlosc` to zmienna przechowująca liczby całkowite, `sTytul` zawiera ciąg znaków, a `bIstnieje` to zmienna logiczna — *przyp. tłum.*

45-procentową skutecznością testowania integracyjnego. Ponadto przeglądy takie zwiększają w dużych projektach ogólną wydajność zespołu. W niektórych przypadkach ograniczają one możliwość wystąpienia awarii do 80 – 90%, a także zapewniają 10 – 25% wzrost wydajności aplikacji (McConnell, 1993). Przegląd powinien być przeprowadzany zarówno podczas testowania, jak i podczas implementacji. Przegląd projektu umożliwia wykrycie jego wad w momencie, gdy ich usuwanie jest najprostsze i najtańsze.

Przegląd może być realizowany na kilka sposobów. Niektóre z nich są formalną inspekcją kodu inne przeglądem ogólnym lub czytaniem kodu. W przypadku formalnej inspekcji kodu kilku członków zespołu spotyka się w celu odszukania błędów. Prowadzący spotkanie powinien dbać o postępy w pracy i pilnować, aby jedynym tematem była identyfikacja błędów. Formalna inspekcja kodu nie powinna dotyczyć zasadności stosowanych rozwiązań. W przypadku przeglądu ogólnego grupa programistów prowadzi nieformalną dyskusję na temat kodu, zadając pytania. Głównym celem spotkania jest identyfikacja błędów, a nie ustalenie sposobów ich usunięcia. Czytanie kodu poświęcone jest jedynie udoskonalaniu istniejącego kodu programu. Zwykle część kodu jest przekazywana dwóm lub więcej osobom, które, pracując niezależnie, identyfikują błędy. Wynik ich pracy jest przekazywany autorowi kodu.

Wybór metody przeglądu kodu zależy od rozmiaru i sposobu organizacji projektu. Jeśli pracujesz sam lub w małym zespole, przegląd metodą czytania kodu możesz zlecić osobie trzeciej. Niezależnie od rodzaju wybranej metody, przeprowadzenie przeglądu kodu jest najbardziej efektywną metodą identyfikacji problemów w projekcie bądź implementacji.

Testowanie

Podobnie jak w przypadku innych projektów, testowanie aplikacji WWW powinno obejmować różne poziomy aplikacji: testowanie funkcji, testowanie modułów oraz testowanie integracyjne. Testowanie każdego modułu powinno być odpowiednio zaplanowane. Testy stanowią zbiór określonych oczekiwań i wymagań.

Testowanie niewielkich projektów może polegać na sprawdzeniu funkcjonowania aplikacji w kilku przypadkach jej wykorzystania. W przypadku tworzenia większych projektów często zatrudniani są specjaliści, którzy zajmują się jedynie testowaniem, jednak każdy z programistów powinien być odpowiedzialny za dostarczenie testerom tylu danych, aby ich praca była efektywna. Każdy twórca kodu powinien również być odpowiedzialny za testowanie swoich modułów na poziomie funkcji lub strony.

Uruchamianie

Uruchamianie jest ostatnim etapem w procesie tworzenia aplikacji, ponieważ w momencie, gdy następuje uruchamianie, powinny być ukończone procesy projektowania, programowania i część testowania. Uruchamianie może być przeprowadzane w każdej fazie testowania jako część tego procesu. Wszystkie zmiany wprowadzone do kodu w trakcie procesu uruchamiania powinny zostać ponownie przetestowane na wszystkich poziomach testowania, ponieważ zmiany te mogą być przyczyną powstania nowych błędów.

Uruchamianie powinno być gruntownym procesem zmierzającym do zidentyfikowania źródeł ewentualnych problemów. Każdy programista biorący udział w testowaniu powinien znać źródło problemu w momencie, gdy stwierdza się, że problem został usunięty. Znalezienie źródła problemu powinno owocować stworzeniem kompletnego rozwiązania, a nie próbą obejścia problemu. W niektórych przypadkach tymczasowe rozwiązania mogą być wystarczające, ale muszą zostać odpowiednio udokumentowane. W trakcie rozwiązywania problemów należy uwzględniać priorytety.

Nie sposób w tej książce, poświęconej głównie PHP, przedstawić wszelkich zasad inżynierii programowania. Programista winien pamiętać, że zasady inżynierii programowania obowiązują we wszystkich aplikacjach, czyli również w aplikacjach WWW, a ich przestrzeganie ułatwia uruchamianie i późniejsze administrowanie aplikacją. W następnej części rozdziału przedstawione są techniki i narzędzia specyficzne dla aplikacji PHP.

Programowanie defensywne

Zanim zaczniesz uruchamiać program, powinieneś podjąć czynności prowadzące do ograniczenia ilości błędów w kodzie. Taki sposób programowania jest nazywany *programowaniem defensywnym*. Polega ono na odpowiednim komentowaniu błędów, a także wewnętrznym sprawdzaniu stanu procedur w trakcie programowania. Każdy programista może komentować błędy na swój sposób, jednak komentarze te powinny być zgodne z ogólnym standardem. Programiści powinni opisywać przeznaczenie funkcji, klasy lub dołączanego pliku oraz zawsze komentować niejasne fragmenty kodu.

Do sprawdzania stanu funkcji PHP, tak jak wiele języków wysokiego poziomu, stosuje funkcję `assert()`. Funkcja `assert()` oblicza wartość przekazanego parametru i podejmuje określone akcje w przypadku, gdy jego wartość wynosi `False`. W PHP do funkcji `assert()` można przekazać zarówno ciąg, jak i wartość Boolean. Jeżeli przekazany został ciąg, jest on wykonywany jako blok kodu PHP. Opcje asercji w pliku *php.ini* (`assert.active`, `assert.warning`, `assert.bail`, `assert.callback` i `assert.quiet_eval`) lub opcje przekazane jako parametr wywołania funkcji `assert_options()` definiują akcję, jaką podejmuje funkcja `assert()`. W tabeli 10.1 wyszczególnione są różne opcje asercji.

Tabela 10.1. *Opcje asercji i ich opis*

Opcja	Domyślnie	Opis
`assert_active`	`1`	Włącza wykonywanie funkcji `assert()`
`assert_warning`	`1`	Wyświetla ostrzeżenie PHP przy każdej nieudanej asercji
`assert_bail`	`0`	Kończy wykonanie w przypadku nieudanej asercji
`assert_quiet_eval`	`0`	Wyłącza raportowanie błędów w czasie obliczenia wyrażenia asercji
`assert_callback`	`(null)`	Nazwa funkcji użytkownika wykonywanej w przypadku nieudanej asercji

Funkcja `assert()` jest zaprojektowana jedynie do wykorzystywania w czasie tworzenia programu i nie powinna być używana w czasie zwykłej pracy aplikacji. Aplikacja powinna działać identycznie z wywołaniami funkcji `assert()`, jak i bez nich. Skrypt z wydruku 10.1 przedstawia przykład wykorzystania funkcji `assert()` do kontrolowania poprawności parametrów wejściowych.

Wydruk 10.1. *Wykorzystanie funkcji assert() do kontrolowania poprawności parametrów wejściowych*

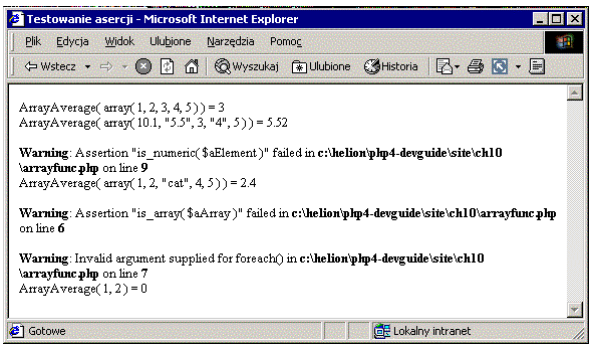
```
<?php
  function ArrayAverage( $aArray )
  {
    $aTotal = 0;
    // Apostrofy ozanczają ciąg nie interpretowany przez PHP
    assert( 'is_array( $aArray )' );
    foreach( $aArray as $aElement )
    {
      assert( 'is_numeric( $aElement )' );
      $aTotal += $aElement;
    }
    return ( $aTotal / count( $aArray ) );
  }
?>
```

Umieszczona w powyższym skrypcie funkcja `ArrayAverage()` spodziewa się jako parametru tablicy wartości numerycznych (liczb lub ciągów zawierających liczby) i zwraca średnią z wartości w tablicy. Wyrażenie `assert()` jest wykorzystywane do kontrolowania poprawności parametru przekazanego do funkcji, a później do sprawdzania, czy tablica zawiera wartości numeryczne. Do `assert()` można przekazać ciąg, który jest wykonywany jako kod PHP, więc jeżeli wykorzystywane są zmienne, należy zapewnić, że PHP nie podstawi zbyt wcześnie wartości zmiennej w miejsce jej nazwy. Aby tego uniknąć, należy używać ciągów w apostrofach. Testowy skrypt dla funkcji z wydruku 10.1 jest zamieszczony na wydruku 10.2.

Wydruk 10.2. *Wykorzystanie funkcji ArrayAverage()*

```
<?php
  include( "./arrayfunc.php" );
?>
<html>
<head>
  <title>Testowanie asercji</title>
</head>
<body>
  <?php
    $aResult = ArrayAverage( array( 1, 2, 3, 4, 5 ) );
    print( "ArrayAverage( array( 1, 2, 3, 4, 5 ) ) = $aResult<br>" );
    $aResult = ArrayAverage( array( 10.1, "5.5", 3, "4", 5 ) );
    print( "ArrayAverage( array( 10.1, \"5.5\", 3, \"4\", 5 ) ) = $aResult<br>" );
    $aResult = ArrayAverage( array( 1, 2, "cat", 4, 5 ) );
    print( "ArrayAverage( array( 1, 2, \"cat\", 4, 5 ) ) = $aResult<br>" );
    $aResult = ArrayAverage( 1, 2 );
    print( "ArrayAverage( 1, 2 ) = $aResult<br>" );
  ?>
</body>
</html>
```

Testowy skrypt wywołuje funkcję `ArrayAverage()` cztery razy. Najpierw dwukrotnie przekazywane są właściwe wartości, a następnie dwukrotnie skrypt przekazuje wartości nieprawidłowe. Wynik działania skryptu jest przedstawiony na rysunku 10.1. Ponieważ PHP wewnętrznie wymusza typy zmiennych, wywołanie `ArrayAverage(array(1, 2, "cat", 4, 5))` jest realizowane bez żadnych ostrzeżeń (poza generowanymi przez asercje).



Rysunek 10.1. Testowanie funkcji ArrayAverage()

Podejmowane przez funkcję `assert()` działania zależą od ustawień asercji. W poprzednim przykładzie wykorzystane zostały domyślne ustawienia asercji. Asercje mają tę zaletę, że gdy `assert.active` jest ustawione na `False`, przestają działać. Aby zmienić tę opcję, należy albo wywołać funkcję `assert_options( ASSERT_ACTIVE, False)`, albo odpowiednio ustawić dyrektywę konfiguracji. Wykorzystując opcję konfiguracji, można instalować aplikację w środowisku z wyłączonymi asercjami, a pracować w takim, w którym asercje są aktywne.

Istnieje jeszcze jedna funkcja pomagająca w programowaniu defensywnym, `error_reporting()`. Funkcja ta jako argumentu wymaga liczby całkowitej określającej poziom raportowania błędów. Argument ten jest traktowany jako maska bitowa, więc można podać zestaw kilku ustawień. PHP posiada zestaw stałych używanych razem z tą funkcją. Są one wyszczególnione poniżej.

Wartość	Nazwa
1	E_ERROR
2	E_WARNING
4	E_PARSE
8	E_NOTICE
16	E_CORE_ERROR
32	E_CORE_WARNING
64	E_COMPILE_ERROR
128	E_COMPILE_WARNING
256	E_USER_ERROR
512	E_USER_WARNING
1024	E_USER_NOTICE

Istnieje również dodatkowa stała, E_ALL, która uaktywnia wszystkie informacje o błędach. W trakcie tworzenia aplikacji poziom raportowania błędów należy ustawić na E_ALL, co spowoduje wyświetlanie wszystkich informacji o błędach w kodzie. Ustawienie to jest również przydatne w trakcie dołączania do aplikacji zewnętrznej biblioteki. Przykładowy kod zamieszczony na wydruku 10.3 generuje ostrzeżenia w przypadku ustawienia maksymalnego poziomu raportowania błędów. Przy standardowych ustawieniach skrypt nie wyświetla żadnego komunikatu.

Wydruk 10.3. *Przykład wykorzystania funkcji error_reporting()*

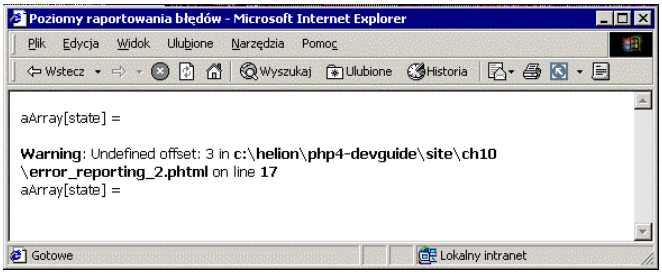
```
<html>
<head>
  <title>Poziomy raportowania błędów</title>
</head>
<body>
<?php
  $aArray = array( "state" => "Idaho", "county" => "Madison",
                   "city" => "Rexburg", "country" => "US" );
  // Standardowy poziom raportowania błędów
  print( "aArray[state] = " . $aArray[state] . "<br>" );
  error_reporting( E_ALL );
  print( "aArray[state] = " . $aArray[state] . "<br>" );
?>
</body>
</html>
```

Wyniki działania powyższego skryptu są przedstawione na rysunku 10.2. Jak można zauważyć, ustawienie bardziej restrykcyjnego poziomu raportowania błędów może zaowocować wykryciem błędów w kodzie, które mogą być źródłem wielu problemów w procesie tworzenia aplikacji. W tym przypadku problem może wydawać się błahy, ale jeżeli zdefiniujemy stałą o nazwie state reprezentującą stan działania aplikacji, problem nabierze znaczenia. Na wydruku 10.4 przedstawiony jest skrypt z taką właśnie stałą. Wyniki jego działania są widoczne na rysunku 10.3.

Wydruk 10.4. *Drugi przykład wykorzystania funkcji error_reporting() oraz stałej state*

```
<?php
    // Stan działania aplikacji
    define( state, 3 );
    // Pozostały kod używający stałej state
?>
<html>
<head>
  <title>Poziomy raportowania błędów</title>
</head>
<body>
<?php
  $aArray = array( "state" => "Idaho", "county" => "Madison",
                   "city" => "Rexburg", "country" => "US" );
  // Domyślny poziom raportowania
  print( "aArray[state] = " . $aArray[state] . "<br>" );
  error_reporting( E_ALL );
  print( "aArray[state] = " . $aArray[state] . "<br>" );
?>
</body>
</html>
```

Rysunek 10.2. Wynik działania skryptu error_reporting()



Rysunek 10.3. Wynik działania drugiego skryptu error_reporting()

W obu przykładach, gdy poziom raportowania błędów był ustawiony na poziom standardowy, nie były generowane żadne ostrzeżenia i program działał niemal bezbłędnie. Jednak po dodaniu stałej (w drugim przykładzie) świetnie działający program nagle przestał działać. Problem taki może być bardzo trudny do zidentyfikowania, jeżeli stała byłaby zdefiniowana w dołączanym pliku. Jak widać, warto ustawiać poziom raportowania błędów na maksimum. Wszystkie wyświetlane ostrzeżenia powinny zostać zlikwidowane, aby uniknąć występowania błędów w przyszłości.

W zależności od specyfiki Twojego środowiska pracy, możesz ustawić poziom raportowania błędów na maksymalny w trakcie tworzenia aplikacji i na minimalny na serwerze produkcyjnym. Takie ustawienie powoduje, że w przeglądarkach użytkowników nie pojawiają się ostrzeżenia i komunikaty błędów. Uważam jednak, że najlepiej ustawić na serwerze produkcyjnym poziom raportowania na maksimum, kierując strumień błędów do pliku dziennika. Można to zrealizować, ustawiając zmienne konfiguracji `display_errors`, `log_errors` i `error_log` na, odpowiednio: `Off`, `On` i `stderr`. Powoduje to, że PHP nie wyświetla błędów w przeglądarce, ale zapisuje je do pliku `stderr`. Dla serwera Apache plikiem `stderr` jest dziennik błędów. Jeżeli chcesz, możesz skorzystać z innej lokalizacji dziennika.

Gdy zostanie wykonany skrypt z wydruku 10.3 po skonfigurowaniu PGP w sposób przedstawiony powyżej, na ekranie nie pojawią się ostrzeżenia, a w pliku dziennika błędów Apache znajdą się następujące pozycje:

```
[06-Dec-2001 20:53:22] PHP Warning:  Undefined offset:  3 in c:\helion\php4-
devguide\site\ch10\error_reporting_2.phtml on line 17
[06-Dec-2001 20:54:03] PHP Warning:  Use of undefined constant state - assumed 'state' in
c:\helion\php4-devguide\site\ch10\error_reporting.phtml on line 12
```

```
[06-Dec-2001 20:54:45] PHP Warning:  Use of undefined constant state - assumed 'state' in
c:\helion\php4-devguide\site\ch10\error_reporting.phtml on line 12
[06-Dec-2001 20:54:49] PHP Warning:  Undefined offset:  3 in c:\helion\php4-
devguide\site\ch10\error_reporting_2.phtml on line 17
[06-Dec-2001 20:54:51] PHP Warning:  Undefined offset:  3 in c:\helion\php4-
devguide\site\ch10\error_reporting_2.phtml on line 17
[06-Dec-2001 20:54:53] PHP Warning:  Use of undefined constant state - assumed 'state' in
c:\helion\php4-devguide\site\ch10\error_reporting.phtml on line 12
```

Następną czynnością realizowaną w ramach programowania defensywnego może być napisanie własnego mechanizmu rejestrującego. W dowolnych miejscach aplikacji można sprawdzać stan niektórych funkcji lub raportować błędy wewnętrzne i kontynuować pracę. PHP udostępnia funkcję `error_log()`, przy pomocy której można dodawać własne zapisy do pliku śladu aplikacji. Prototyp funkcji `error_log()` wygląda następująco:

```
int error_log(string komunikat, int typ [, string cel [, string naglowki]])
```

Pierwszy parametr, `komunikat`, zawiera zapisywane dane. Drugi z parametrów określa miejsce, w którym zostanie zapisany komunikat. Lista prawidłowych wartości parametru `typ` znajduje się w tabeli 10.2.

Tabela 10.2. *Wartości parametru typ*

Wartość	Opis
0	Parametr `komunikat` jest wysyłany do systemowego mechanizmu rejestrowania PHP przy użyciu mechanizmu rejestrowania zapewnianego przez system operacyjny lub pliku — w zależności od ustawienia zmiennej konfiguracji `error_log`
1	Komunikat jest wysyłany pocztą elektroniczną na adres podany w parametrze `cel`. Jedynie ten typ zapisu wykorzystuje parametr `naglowki`. Do obsługi tego typu komunikatu wykorzystywana jest ta sama funkcja wewnętrzna, co w funkcji `mail()`
2	Komunikat jest wysyłany poprzez połączenie PHP używane do uruchamiania zdalnego. Opcja ta jest dostępna jedynie w przypadku, gdy jest włączone uruchamianie zdalne. W tym przypadku parametr `cel` określa nazwę komputera lub adres IP oraz, opcjonalnie, numer portu używanego do odbierania informacji uruchamiania
3	Komunikat jest dołączany do końca pliku o nazwie `cel`

W czasie pisania książki typ 2 nie był dostępny. Kod źródłowy zawierał komentarz informujący, że zdalne uruchamianie nie jest jeszcze dostępne. Inne typy komunikatów działają w sposób opisany w tabeli. Skrypt zamieszczony na wydruku 10.5 stanowi przykład wykorzystania funkcji `error_log()`.

Wydruk 10.5. *Wykorzystanie funkcji error_log()*

```
<html>
<head>
  <title>Rejestrowanie błędów</title>
</head>
<body>
  <?php
    // Zapisanie błędu do dziennika systemowego
```

```
    error_log( "MÓJ BŁĄD: wystąpił błąd!", 0 );
    // Wysłanie błędu pocztą e-mail
    error_log( "MÓJ BŁĄD: wystąpił błąd!", 1, "app_errors@intechra.net",
               "From: error_logger@myhost.com\r\n" );
    // Zapisanie błędu w pliku śladu aplikacji
    error_log( "MÓJ BŁĄD: wystąpił błąd!", 3, "/tmp/error.log" );
  ?>
    Wystąpiły błędy.
</body>
</html>
```

Pierwsze wywołanie funkcji `error_log()` zapisuje komunikat błędu w systemowym dzienniku błędów. W powyższym przykładzie błąd ten wysyłany jest do dziennika błędów Apache. Drugie wywołanie skutkuje wysłaniem poczty elektronicznej do odbiorcy określonego w parametrze `cel`. Informacje zawarte w dodatkowych nagłówkach są utworzone na podstawie tych samych zasad, co w przypadku funkcji `mail()`. Ostatnie wywołanie powoduje zapisanie komunikatu błędu do pliku, w tym przypadku */tmp/error.log*.

Wykorzystanie tego mechanizmu nie jest w dosłownym znaczeniu uruchamianiem, ale jego zastosowanie może znacznie skrócić i ułatwić właściwe uruchamianie, umożliwiając wyeliminowanie niektórych błędów i przeoczeń już w fazie programowania. Jak wcześniej wspomniałem, ograniczenie liczby błędów w kodzie powinno być priorytetem dla każdego programisty.

Dostosowanie mechanizmu obsługi błędów do potrzeb aplikacji

Tak jak niemal każdy element PHP, mechanizm obsługi błędów można dostosować do własnych potrzeb, aby spełniał wymagania aplikacji. Funkcja `error_log()` umożliwia skonstruowanie prostego mechanizmu rejestrowania własnych błędów występujących w aplikacji, ale nie umożliwia obsługi błędów generowanych przez PHP. Nie pozwala również na przechwytywanie komunikatów generowanych przez funkcje `assert()`. Na szczęście w PHP takie przypadki można obsługiwać w inny sposób.

Funkcja `set_error_handler()` pozwala zarejestrować funkcję w PHP, która będzie wywoływana za każdym razem, gdy zostanie wygenerowany komunikat błędu. Funkcja `set_error_handler()` wymaga podania jednego argumentu — nazwy funkcji obsługi błędów. Prototyp takiej funkcji wygląda następująco:

```
function ErrorCallBack( int nr_bledu, string ciag_bledu, string nazwa_skryptu, int
nr_lini, array kontekst)
```

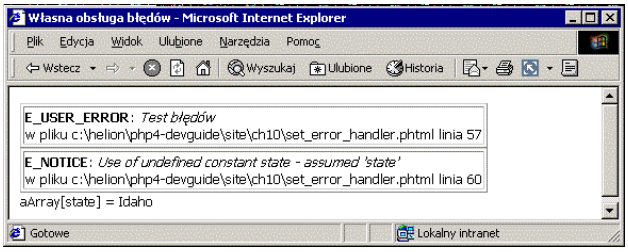
Sposób rejestrowania i wykorzystania funkcji obsługi błędów jest przedstawiony w przykładowym skrypcie na wydruku 10.6.

Wydruk 10.6. *Wykorzystanie funkcji set_error_handler()*

```
<?php
  function myErrorHandler( $aErrorNo, $aErrorStr, $aFile, $aLine, $aContext)
  {
    switch ( $aErrorNo )
    {
      case E_ERROR:
          $aErrorType = "E_ERROR"; // nie powinien wystąpić
          break;
      case E_WARNING:
          $aErrorType = "E_WARNING";
          break;
      case E_PARSE:
          $aErrorType = "E_PARSE"; // nie powinien wystąpić
          break;
      case E_NOTICE:
          $aErrorType = "E_NOTICE";
          break;
      case E_CORE_ERROR:
          $aErrorType = "E_CORE_ERROR"; // nie powinien wystąpić
          break;
      case E_CORE_WARNING:
          $aErrorType = "E_CORE_WARNING"; // nie powinien wystąpić
          break;
      case E_COMPILE_ERROR:
          $aErrorType = "E_COMPILE_ERROR"; // nie powinien wystąpić
          break;
      case E_COMPILE_WARNING:
          $aErrorType = "E_COMPILE_WARNING"; // nie powinien wystąpić
          break;
      case E_USER_ERROR:
          $aErrorType = "E_USER_ERROR";
          break;
      case E_USER_WARNING:
          $aErrorType = "E_USER_WARNING";
          break;
      case E_USER_NOTICE:
          $aErrorType = "E_USER_NOTICE";
          break;
      default:
          $aErrorType = "UNKNOWN ERROR TYPE";
          break;
    }
    print( "<table border=\"1\"><tr><td>" );
    print( "<b>$aErrorType</b>: <i>$aErrorStr</i><br>" );
    print( "w pliku $aFile linia $aLine<br>" );
    print( "</td></tr></table>" );
  }
  set_error_handler( "myErrorHandler" );
  error_reporting( E_ALL );
?>
<html>
<head>
  <title>Własna obsługa błędów</title>
</head>
<body>
  <?php
```

```
    trigger_error( "Test błędów", E_USER_ERROR );
    $aArray = array( "state" => "Idaho", "county" => "Madison",
                     "city" => "Rexburg", "country" => "US" );
    print( "aArray[state] = " . $aArray[state] . "<br>" );
  ?>
</body>
</html>
```

W powyższym skrypcie została zdefiniowana funkcja obsługi błędów `myErrorHandler()`, która wyświetla komunikaty błędów w obramowanej tabeli zawierającej jedną komórkę, co pomaga w odróżnieniu komunikatu błędu od reszty kodu HTML. Po zainstalowaniu funkcji obsługi skrypt powoduje dwa błędy. Pierwszy jest generowany przy użyciu funkcji PHP `trigger_error()`. Drugi błąd (ostrzeżenie) jest identyczny z błędem przedstawionym na wydruku 10.3. Na rysunku 10.4 przedstawiony jest wynik działania skryptu.



Rysunek 10.4. Działanie funkcji set_error_handler()

Przy stosowaniu funkcji `set_error_handler()` należy pamiętać o tym, że PHP nie przekazuje do funkcji obsługi błędów typu `E_ERROR`, `E_PARSE`, `E_CORE_ERROR`, `E_CORE_WARNING`, `E_COMPILE_ERROR` oraz `E_COMPILE_WARNING`. Obsługa tego typu błędów przez użytkownika nie jest bezpieczna. Z tego powodu w kodzie funkcji obsługi błędów z wydruku 10.6 pojawiły się komentarze „nie powinien wystąpić".

Uwaga na temat funkcji set_error_handler()

Funkcja `set_error_handler()` jest dostępna w PHP od wersji 4.0.1. Dodatkowo, funkcja użyta w tym przykładzie posiada pięć parametrów, w tym nazwę skryptu, numer linii i dane kontekstu. Parametry te są dostępne dopiero w wersji 4.0.2. Wcześniejsze wersje miały tylko dwa parametry: typ komunikatu i komunikat.

W skrypcie z wydruku 10.6 nie są wykorzystane dane na temat kontekstu. Będą one opisane w następnej części rozdziału. Dane te zawierają nazwy i wartości zmiennych istniejących w skrypcie w momencie wystąpienia błędu.

Również funkcja `assert()` umożliwia zdefiniowanie wywoływanej funkcji. Aby to zrealizować, należy zastosować funkcję `assert_options()`. Funkcja obsługująca nieudane asercje jest zdefiniowana w następujący sposób:

```
function AssertCallback ($NazwaPliku, $NrLinii, $Asercja )
```

Uwaga na temat funkcji assert_options()

W PHP do wersji 4.0.2 włącznie w funkcji `assert_options()` był obecny drobny błąd. Występował on w przypadku wywołania funkcji w postaci `assert_options(ASSERT_CALLBACK)` w celu odczytania ustawionej funkcji obsługi. Mimo że w dokumentacji napisano, że wywołanie takie zwróci jedynie nazwę bieżącej funkcji obsługi, to dodatkowo, oprócz zwracania nazwy, usuwane było bieżące ustawienie funkcji obsługi. Dlatego jeżeli chcesz użyć funkcji `assert()` z funkcją obsługi, upewnij się, że nie jest wywoływana później funkcja `assert_options()` w celu sprawdzenia nazwy zarejestrowanej funkcji. Błąd ten został zauważony i poprawiony w PHP od wersji 4.0.2.

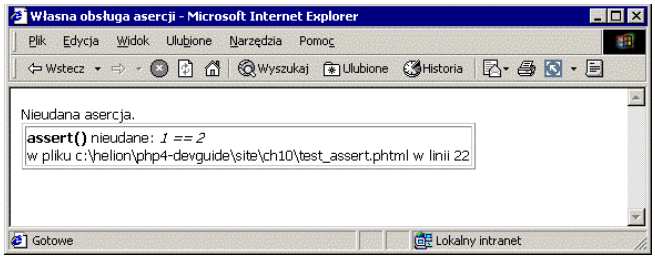
Skrypt zamieszczony na wydruku 10.7 przedstawia przykład zdefiniowania i użycia funkcji wywoływanej przez `assert()`.

Wydruk 10.7. *Wykorzystanie funkcji wywoływanej przez ASSERT_CALLBACK*

```
<?php
  error_reporting( E_ALL );
  function MyACallback( $aFileName, $aLineNum, $aAssertion )
  {
    print( "<table border=\"1\"><tr><td>" );
    print( "<b>assert()</b> nieudane: <i>$aAssertion</i><br>" );
    print( "w pliku $aFileName w linii $aLineNum<br>" );
    print( "</td></tr></table>" );
  }
  // Zarejestrowanie funkcji obsługi
  assert_options( ASSERT_CALLBACK, "MyACallback" );
  // Wyłączenie wyświetlania zwykłych ostrzeżeń
  assert_options( ASSERT_WARNING, 0 );
?>
<html>
<head>
  <title>Własna obsługa asercji</title>
</head>
<body>
  Nieudana asercja.
  <?php
    assert( "1 == 2" );
  ?>
</body>
</html>
```

Kod z powyższego wydruku jest podobny do tego z wydruku 10.6. Wywołanie asercji powoduje wyświetlenie jednokomórkowej tabeli. Jeżeli nie zostanie zastosowana opcja `ASSERT_WARNING`, oprócz informacji zdefiniowanych przez użytkownika wyświetlony zostanie standardowy komunikat PHP. Na rysunku 10.5 przedstawiony jest wynik działania skryptu z wydruku 10.7.

Mechanizm obsługi błędów w PHP jest bardzo elastyczny. Ułatwia to uruchamianie i późniejsze utrzymywanie aplikacji. W następnej części rozdziału połączymy przedstawione dotychczas techniki, co w efekcie ułatwi uruchamianie programów w całym cyklu produkcyjnym.



Rysunek 10.5. Efekt zastosowania funkcji zdefiniowanej dla assert()

Zaawansowana obsługa błędów

Po omówieniu technik obsługi błędów możemy rozpocząć tworzenie ogólnego narzędzia do obsługi błędów. Motywacją do napisania tego fragmentu była nieobecność w PHP narzędzi, które automatycznie łączyłyby różne typy danych o błędach. Oprócz tego, PHP 3 posiadał możliwość zdalnego uruchamiania, która nie została przeniesiona do PHP 4. Mechanizm ten umożliwiał przesyłanie danych za pomocą protokołu TCP/IP do innego komputera. Opcja ta prawdopodobnie niedługo się pojawi, ale na razie muszą nam wystarczyć podstawowe techniki obsługi błędów przedstawione w tym rozdziale.

Przykład przedstawiony w tej części jest niezwykle długi, dlatego zostanie omówiony we fragmentach. Moduł ten, określony mianem *MyDebug*, znajduje się w jednym pliku, *MyDebug.php*. Jest on utworzony w taki sposób, że może być bez przeszkód dołączony do dowolnego skryptu PHP. Po dołączeniu pliku wykonywany jest kod zawarty na wydruku 10.8.

Wydruk 10.8. *Dołączanie modułu MyDebug*

```
getenv( "MYDEBUG_CONFIG" );
if ( $aMyDebugType & MYDEBUG_DISPLAYFILE )
{
    // Jeżeli nie można zapisać do pliku śladu,
    // poinformuj o tym użytkownika i wyłącz zapis do pliku.
    if ( CheckFileSanity( $aMyDebugFile ) == False )
    {
        error_log( "MyDebug nie udało się otworzyć pliku $aMyDebugFile", 0 );
        $aMyDebugType &= ~MYDEBUG_DISPLAYFILE;
    }
}

if ( $aMyDebugType & MYDEBUG_DISPLAYFILE )
{
    $aFileHandle = fopen( $aMyDebugFile, "a" );
}

if ( $aMyDebugType & MYDEBUG_DISPLAYIP )
{
    $aSocketHandle = fsockopen( "udp://$aMyDebugIP", $aMyDebugPort );
}
```

```
//  Teraz rejestrujemy funkcje obsługi i funkcje porządkujące.
set_error_handler( "MyErrHandler" );
assert_options( ASSERT_CALLBACK, "MyAssertHandler" );
assert_options( ASSERT_WARNING, 0 );
register_shutdown_function( "MyDebugShutdown" );
```

Pierwsza linia skryptu z wydruku 10.8 powoduje przetworzenie ciągu konfiguracji *MyDebug*, który jest przechowywany w zmiennej środowiska serwera. Na przykład, plik konfiguracyjny Apache może zawierać kod podobny do następującego:

```
SetEnv MYDEBUG_CONFIG FILE=/tmp/mydebug.log;
MAIL=mydebug@intechra.net;IP=myserver.com:5400;
```

Ta opcja konfiguracji jest specyficzna dla modułu *MyDebug* i nie jest dostępna jako standardowa część Apache czy PHP. Jak widać, można ustawiać zmienne środowiska poprzez pliki konfiguracyjne serwera WWW. Zmienne te są dostępne w kodzie PHP. Zmienna `MYDEBUG_CONFIG` definiuje miejsca, w których *MyDebug* będzie zapisywał błędy. W tym przypadku moduł będzie zapisywał błędy do pliku (*/tmp/mydebug.log*), wysyłał na adres e-mailowy (*mydebug@intechra.net*) oraz do gniazda UDP (*myserver.com:5400*). Zwykle wybiera się jedną z metod zapisu błędów, ale moduł *MyDebug* umożliwia stosowanie wielu miejsc zapisu błędów naraz. Funkcja `ParseConfg()` analizuje ciąg konfiguracji i ustawia odpowiednie zmienne globalne.

Po przeanalizowaniu ciągu konfiguracyjnego sprawdzane są wszystkie pliki używane do zapisywania danych o błędach. Jeżeli moduł *MyDebug* nie może zapisać danych do pliku, zapisywanie do niego jest wyłączane. Następnie otwierane są wszystkie potrzebne pliki i gniazda. Gniazdo jest otwierane z wykorzystaniem UDP, co nie wymaga istnienia procesu nasłuchu. Własność ta jest użyteczna szczególnie wtedy, gdy zapisywanie jest aktywne na serwerze produkcyjnym, ale nie zawsze dostępny jest proces nasłuchu.

Następnie moduł *MyDebug* rejestruje funkcje obsługi błędów i asercji. Ostatnią operacją jest zarejestrowanie funkcji wywoływanej po zakończeniu działania programu, co umożliwia eleganckie zakończenie działania modułu. Wydruk 10.9 przedstawia kod funkcji kończącej działanie programu.

Wydruk 10.9. *Kod funkcji kończącej działanie programu*

```
function MyDebugShutdown( )
{
  global $aFileHandle, $aSocketHandle, $aMyDebugType;
  if ( $aMyDebugType & MYDEBUG_DISPLAYFILE )
  {
    fclose( $aFileHandle );
  }

  if ( $aMyDebugType & MYDEBUG_DISPLAYIP )
  {
    fclose( $aSocketHandle );
  }
}
```

Funkcja ta zamyka wymagane pliki oraz gniazdo sieciowe. Funkcję kończącą może zarejestrować dowolny skrypt. PHP pozwala na rejestrowanie wielu funkcji kończących, które są wykonywane w czasie kończenia pracy skryptu. Choć ten fakt nie jest odnotowany w dokumentacji, funkcje kończące są wywoływane w tej samej kolejności, w jakiej zostały zarejestrowane. Ponieważ kolejność ta nie została udokumentowana, Twoje skrypty nie powinny polegać na kolejności wykonywania funkcji kończących. Jeżeli zależy Ci na wykonywaniu ich w odpowiednim porządku, musisz sam sprawdzić kolejność ich wywoływania. Można przekazać dodatkowe argumenty do funkcji `register_shutdown_function()`. PHP przekaże je do funkcji końcowej. Należy również pamiętać, że w funkcji końcowej nie można wysyłać żadnych danych do przeglądarki. Po skonfigurowaniu moduł *MyDebug* obsługuje błędy za pomocą funkcji z wydruku 10.10.

Wydruk 10.10. *Funkcje obsługi błędów*

```
//  Funkcja obsługi ustawiana przez set_error_handler()
function MyErrHandler( $aErrorNo, $aErrorStr, $aFile, $aLine, $aContext)
{
  MyDebug( $aErrorStr, $aFile, $aLine, MYDEBUG_ERRCALLBACK,
           $aErrorNo, $aContext );
}

//  Funkcja obsługi dla funcji assert_options()
function MyAssertHandler( $aFileName, $aLineNum, $aAssertion )
{
  MyDebug( "asercja( $aAssertion ) nieudana", $aFileName,
           $aLineNum, MYDEBUG_ASSERTCALLBACK );
}
```

Obie funkcje przekazują parametry do głównej funkcji obsługi błędów, która może również zostać wywołana bezpośrednio z poziomu skryptu. Główna funkcja obsługi błędów to przedstawiona na wydruku 10.11 funkcja `MyDebug()`.

Wydruk 10.11. *Funkcja MyDebug()*

```
//  Funkcja MyDebug jest główną funkcją obsługi
function MyDebug( $aMessage, $aFile, $aLine,
                  $aCallType = MY_DEBUG_INTERNAL, $aErrType = 0,
                  $aErrContext = array() )
{
  global $aMyDebugType;

  for ( $aDisplayType = MYDEBUG_DISPLAYFILE;
        $aDisplayType <= MYDEBUG_DISPLAYIP;
        $aDisplayType++ )
  {
    if ( $aDisplayType & $aMyDebugType )
    {
      $aType      = FormatType( $aCallType, $aDisplayType );
      $aMessage   = FormatMsg ( $aCallType, $aMessage, $aErrType, $aDisplayType );

      if ( $aCallType == MYDEBUG_ERRCALLBACK )
      {
          $aContext = FormatContext( $aErrContext, $aDisplayType );
      }
```

```
        else
        {
            $aContext = "";
        }

        MyDebugOutput( $aType, $aMessage, $aFile, $aLine, $aContext, $aDisplayType );
      }
    }
  }
```

Funkcja `MyDebug()` formatuje różne parametry w zależności od typu medium zapisu (plik, e-mail lub TCP/IP). Następnie wywołuje funkcję `MyDebugOutput()` (wydruk 10.12), która wysyła dane do prawidłowego miejsca. Funkcje formatujące z wydruku 10.11 zostaną omówione w dalszej części rozdziału.

Wydruk 10.12. *Funkcja MyDebugOutput()*

```
function MyDebugOutput( $aType, $aMessage, $aFile, $aLine,
                        $aContext, $aDisplayType )
 {
  global $aFileHandle, $aSocketHandle, $aMyDebugEmail;

  switch( $aDisplayType )
  {
    case MYDEBUG_DISPLAYFILE:
      $aMsg = "$aType: '$aMessage' wystąpił w $aFile w lini $aLine. ";
      if ( $aContext != "" )
      {
          $aMsg .= "Dane kontekstu:\n{$aContext}\n";
      }
      else
      {
          $aMsg .= "\n";
      }
      fputs( $aFileHandle, $aMsg );
      break;
    case MYDEBUG_DISPLAYEMAIL:
      $aMsg = "$aType: '$aMessage' wystąpił w $aFile w linii $aLine. ";
      if ( $aContext != "" )
      {
          $aMsg .= "Dane kontekstu:\n{$aContext}\n";
      }
      else
      {
          $aMsg .= "\n";
      }
      mail($aMyDebugEmail, "Raport MyDebug", $aMsg, "From: mydebug@host.com\r\n");
      break;
    case MYDEBUG_DISPLAYIP:
      $aMsg = "$aType|$aMessage|$aFile|$aLine|$aContext^^";
      fputs( $aSocketHandle, $aMsg );
      break;
  }
 }
```

Funkcja `MyDebugOutput()` wysyła dane do właściwych miejsc. Jest ona zaskakująco prosta, jeżeli uwzględnimy efektywność każdej z tych opcji. Każda funkcja formatująca użyta w module *MyDebug* posiada mechanizm zamiany wewnętrznego numeru błędu na postać tekstową, zrozumiałą dla użytkownika. Na przykład funkcja `FormatType()`, przedstawiona na wydruku 10.13, formatuje kod typu błędu.

Wydruk 10.13. *Funkcja FormatType()*

```
/*  Funkcja formatuje typ komunikatu na podstawie tego,
    gdzie będzie on wyświetlony */
function FormatType( $aCallType, $aDisplayType )
{
  switch( $aDisplayType )
  {
    case MYDEBUG_DISPLAYFILE:
    case MYDEBUG_DISPLAYEMAIL:
      switch ( $aCallType )
      {
          case MYDEBUG_INTERNAL:
              return "INTERNAL";
              break;
          case MYDEBUG_ERRCALLBACK:
              return "ERROR CALLBACK";
              break;
          case MYDEBUG_ASSERTCALLBACK:
              return "ASSERT CALLBACK";
              break;
      }
      break;
    case MYDEBUG_DISPLAYIP:
      return $aCallType;
      break;
  }
}
```

Jeżeli dane są wysyłane poprzez TCP/IP, formatowanie nie jest przeprowadzane. Sam numer typu jest wysyłany do zdalnego komputera. W innym wypadku numer typu jest zamieniany na czytelny ciąg. Inne funkcje konwertujące użyte w *MyDebug* działają podobnie. Jedynie funkcja formatująca kontekst błędu, `FormatContext`(), jest wyraźnie odmienna. Funkcja ta jest wywoływana jedynie wtedy, gdy błąd zostanie obsłużony przez funkcję zarejestrowaną za pomocą `set_error_handler()`. Dane kontekstu udostępniane przez PHP zawierają wszystkie zmienne będące w zasięgu w momencie wystąpienia błędu. Dane te są przesyłane w postaci tablicy asocjacyjnej z nazwami zmiennych i ich wartościami. Analiza tych danych wymaga wykorzystania funkcji rekurencyjnej, ponieważ w kontekście mogą znajdować się tablice. Funkcja zamieszczona na wydruku 10.14 analizuje dane kontekstu, przekształcając je w postać czytelną dla człowieka.

Wydruk 10.14. *Analiza danych kontekstu*

```
/*  Funkcja formatuje typ komunikatu na podstawie tego,
    gdzie będzie on wyświetlony. Funkcja ta wykorzystuje
    funkcję rekurencyjną FormatContextR */
function FormatContext( $aErrContext, $aDisplayType )
{
```

```
  // Od tej pory wszystkie wyświetlane typy
  // otrzymują ten sam ciąg kontekstu
  $aString = "";
  $aDelim  = "\n";
  FormatContextR( $aErrContext, $aString, $aDelim );
  return $aString;
}

function FormatContextR( $aErrContext, &$aString, $aDelim )
{
  foreach( $aErrContext as $aVarName => $aVarValue )
  {
    if ( is_array( $aVarValue ) == True )
    {
      $aString .= "$aVarName = array( ";
      FormatContextR( $aVarValue, $aString, "," );
      $aString .= " )$aDelim";
    }
    else
    {
      $aString .= "$aVarName = {$aVarValue}{$aDelim}";
    }
  }
}
```

Funkcja `FormatContext()` ustawia kilka parametrów i wywołuje funkcję rekurencyjną `FormatContextR()`. Funkcja rekurencyjna przegląda tablice zmiennych kontekstu i zapisuje każdą parę nazwa-wartość do wynikowego ciągu. Jeżeli zostanie napotkana tablica, wywoływana jest rekurencyjnie funkcja `FormatContextR()`.

W zależności od miejsca wystąpienia błędu, kontekst lokalny może zawierać sporo danych. Jeżeli błąd wystąpi w głównej części skryptu, w zasięgu znajdą się wszystkie zmienne globalne, w tym zmienne środowiska i zmienne `GET` i `POST`. Wszystkie te zmienne znajdą się w danych kontekstu. Jeżeli błąd wystąpi w funkcji, kontekst będzie zawierał jedynie zmienne lokalne funkcji.

Do skryptu testującego (zobacz wydruk 10.15) dołączyliśmy moduł *MyDebug* oraz ustawiliśmy zmienną konfiguracji na zapisywanie do pliku tekstowego. Po jego uruchomieniu na końcu pliku śladu znalazł się ciąg błędu. Poniższy tekst nie jest całym plikiem, jedynie wynikiem wystąpienia ostatniego błędu w skrypcie:

```
ERROR CALLBACK: 'Typ błędu PHP: E_USER_ERROR - error in sum'
wystąpił w c:\helion\php4-devguide\site\ch10\test_mydebug.phtml w lini 16. Dane kontekstu:
a = 1
b = 2
aArray = array( 0 = spring,1 = summer,2 = autumn,3 = winter, )
```

Wydruk 10.15. *Skrypt testowy*

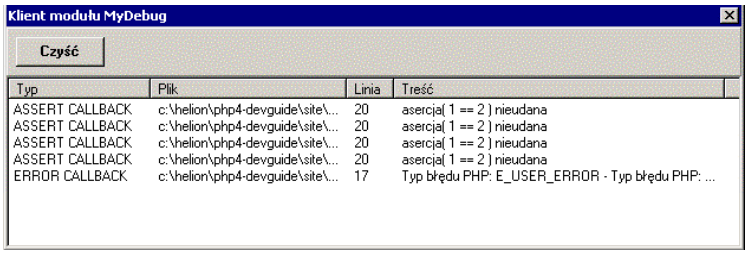
```
<?php
    include_once( "./mydebug.php" );
?>
<html>
<head>
   <title>Test modułu MyDebug</title>
```

```
</head>
<body>
  Nieudana asercja.<br><br>
  <?php
    function sum( $a, $b )
    {
        $aArray = array( "spring", "summer", "autumn", "winter" );
        trigger_error( "error in sum", E_USER_ERROR );
    }

    assert( "1 == 2" );
    trigger_error( "Błąd testowy", E_USER_ERROR );
    $aArray = array( "state" => "Idaho", "county" => "Madison",
                     "city" => "Rexburg", "country" => "US" );
    print( "<br><br>aArray[state] = " . $aArray[state] . "<br>" );
    sum( 1, 2 );
  ?>
</body>
</html>
```

Jedynym zadaniem powyższego skryptu testowego jest generowanie błędów. Wynik działania funkcji `sum()` nie jest nigdzie używany. Jest ona umieszczona w tym skrypcie po to, aby zilustrować, w jaki sposób wywołanie funkcji wpływa na dane kontekstu przekazywane przez PHP. Linie z opisem błędu zamieszczone bezpośrednio przed wydrukiem 10.15 zostały wygenerowane przy wywołaniu funkcji `sum()`.

Elastyczność tego modułu dobrze obrazuje aplikacja Windows, która realizuje proces nasłuchu portu TCP/IP i wyświetla przychodzące dane. Jest to prosta aplikacja Delphi, która odczytuje pakiety UDP przychodzące do portu 5400. Po odczytaniu danych aplikacja formatuje linie i wyświetla je. Na rysunku 10.6 przedstawiona została ta aplikacja po odebraniu kilku komunikatów wygenerowanych przez PHP.



Rysunek 10.6. Aplikacja nasłuchu dla MyDebug

Jednym z powodów popularności języka PHP jest jego niezwykła rozszerzalność. Moduł *MyDebug* jest napisany całkowicie w PHP i ponieważ nie jest kompletny, może być rozbudowywany na wiele sposobów (źródła modułu *MyDebug* są dostępne wraz ze wszystkimi przykładami kodu z tej książki). Na przykład, wykorzystanie poczty elektronicznej do raportowania błędów jest nieefektywne, ale można wykorzystać pocztę elektroniczną do raportowania wyłącznie krytycznych błędów i ostrzeżeń, co pozwoli na wykorzystanie tej opcji w środowisku produkcyjnym. Niezmiernie istotny jest fakt, że wszystkie te opcje są zrealizowane całkowicie w PHP. Nie wszystkie narzędzia programowania dla WWW są tak elastyczne.

Podsumowanie

W tym rozdziale przedstawione zostały techniki programowania defensywnego, których zastosowanie ułatwia późniejsze uruchamianie aplikacji. PHP nie został jeszcze wyposażony w program do uruchamiania skryptów podobny do tych, które są dostępne w wielu nowoczesnych językach programowania. Jednak odrobina pomysłowości i wykorzystanie rozszerzalności PHP pozwala stworzyć świetne narzędzia służące do uruchamiania aplikacji. W rozdziale opisany został jeden z modułów, który zapewnia elastyczną obsługę błędów i może być dowolnie modyfikowany i rozbudowywany.

Bibliografia

Steve McConnell, *Code Complete*, Microsoft Press, Seattle 1993.